J

INTRODUCTION & DICTIONARY

Kenneth E. Iverson

J Version 6.2

Iverson Software Inc.
33 Major Street
Toronto, Ontario
Canada M5S 2K9

ISBN 1-895721-06-7

TABLE OF CONTENTS

1: INTRODUCTION

J is a general-purpose programming language available as shareware on a wide variety of computers. Although it has a simple structure, is treated completely in a thirty-five page dictionary, and is readily learned by anyone familiar with mathematical notions and notation, its distinctive features may make it difficult for anyone familiar with more conventional programming languages.

This book is designed to introduce **J** in a manner that makes it easily accessible to programmers, by emphasizing those aspects that distinguish it from other languages. These include:

1. A mnemonic one- or two-character spelling for primitives.

2. No order-of-execution hierarchy among functions.

3. The systematic use of *ambivalent* functions that, like the minus sign in arithmetic, can denote one function when used with two arguments (*subtraction* in the case of `-`), and another when used with one argument (*negation* in the case of `-`).

4. The adoption of terms from English grammar that better fit the grammar of **J** than do the terms commonly used in mathematics and in programming languages. Thus, a function such as addition is also called a *verb* (because it performs an action), and an entity that modifies a verb (not available in most programming languages) is accordingly called an *adverb*.

5. The systematic use of adverbs and conjunctions to modify verbs, so as to provide a rich set of operations based upon a rather small set of verbs. For example, `+/a` denotes the sum over a list `a`, and `*/a` denotes the product over `a`, and `a */ b` is the multiplication table of `a` and `b`.

6. The treatment of vectors, matrices, and other arrays as single entities.

7. The use of *functional* or *tacit* programming that requires no explicit mention of the arguments of a function (program) being defined, and the simple use of assignment to assign names to functions (as in `sum=. +/` and `mean=. sum % #`).

The following lessons are records of actual **J** sessions, accompanied by commentary that should be read only after studying the corresponding session (and perhaps experimenting with variations on the computer). The lessons should be studied with a **J** system at hand. The reckless reader may go directly to the sample topics on page 31.

2: MNEMONICS

The left side of the page shows an actual computer session with the result of each sentence shown at the left margin. First cover the comments at the right, and then attempt to state in English the meaning of each primitive so as to make clear the relations between related symbols. For example, "`<` is *less than*" and "`<.` is *lesser of* (that is, minimum)". Then uncover the comments and compare with your own.

`   7<5`	Less than
`0`	A zero is interpreted as *false*.
`   7<.5`	Lesser of
`5`	
`   7>5`	Greater than
`1`	A one is *true* (*à la* George Boole)
`   7>.5`	Greater of
`7`	
`   10^3`	Power (*à la* Augustus de Morgan)
`1000`	
`   10^.1000`	Logarithm
`3`	
`   7=5`	Equals
`0`	
`   b=. 5`	Is (*assignment* or *copula*)
`   7<. b`	
`5`	
`   Min=. <.`	`Min` is `<.`
`   power=. ^`	`power` is `^`
`   gt=. >`	`gt` is `>`
`   10 power (5 Min 3)`	
`1000`	

Exercises for all lessons begin on page 45.

Do the exercises for this lesson.

3: AMBIVALENCE

Cover the comments on the right and provide your own.

`   7-5`
`2`
 	
`   -5`
`_5`
 	
`   7%5`
`1.4`
 	
`   %5`
`0.2`
	The function in the sentence `7-5` applies to two arguments to perform subtraction, but in the sentence `-5` it applies to a single argument to perform negation. Adopting from chemistry the term *valence*, we say that the symbol - is *ambivalent*, its effective binding power being determined by context. The ambivalence of - is familiar in arithmetic; it is here extended to other functions.

`   3^2` `9`	
`   ^2` `7.38906`	*Exponential* (that is, `2.71828^2`)
`   a=. i. 5` `   a`	The function *integer* or *integer list*
`0 1 2 3 4`	*List* or *vector*
`   a i. 3 1` `3 1`	The function *index* or *index of*

`   b=. 'Canada'`
`   b i. 'da'`
`4 1`
	Enclosing quotes denote literal characters

`   $ a` `5`	Shape function

`   3 4 $ a`
`0 1 2 3`
`4 0 1 2`
`3 4 0 1`
	Reshape function
	Table or *matrix*

`   3 4 $ b`
`Cana`
`daCa`
`nada`	

`   %a`	Functions apply to lists
`_ 1 0.5 0.333333 0.25`	_ alone denotes *infinity*

Do the exercises for this lesson.

4: VERBS AND ADVERBS

In the sentence `%a` of Lesson 3, the `%` "acts upon" `a` to produce a result, and `%a` is therefore analogous to the notion in English of a *verb* acting upon a *noun* or *pronoun.* We will hereafter adopt the term *verb* instead of (or in addition to) the mathematical term *function* used thus far.

The sentence `+/ 1 2 3 4` is equivalent to `1+2+3+4`; the adverb `/` applies to *its* verb argument `+` to produce a new verb whose argument is `1 2 3 4`, and which is defined by inserting the verb `+` between the items of its argument. Other arguments of the insert adverb are treated similarly:

```
   */b=.2 7 1 8 2 8
1792
   <./b
1
   >./b
8
```

The verb resulting from the application of an adverb may (like a primitive verb) have both monadic and dyadic cases. In the present instance of `/` the dyadic case produces a *table*. For example:

```
   2 3 5 +/ 0 1 2 3
2 3 4 5
3 4 5 6
5 6 7 8
```

The verbs `over=. ({.;}.)@":@,` and `by=. ' '&;@,.@[,.]` can be entered as *utilities* (for use rather than for immediate study), and can clarify the interpretation of *function tables* such as the addition table produced above. For example:

```
   a=. 2 3 5
   b=. 0 1 2 3
   a by b over a +/ b
┌─┬───────┐
│ │0 1 2 3│
├─┼───────┤
│2│2 3 4 5│
│3│3 4 5 6│
│5│5 6 7 8│
└─┴───────┘
```

```
   b by b over b </ b
┌─┬───────┐
│ │0 1 2 3│
├─┼───────┤
│0│0 1 1 1│
│1│0 0 1 1│
│2│0 0 0 1│
│3│0 0 0 0│
└─┴───────┘
```

Due to its two uses, the adverb `/` is often called either *insert* or *table*. Do the exercises for this lesson.

5: PUNCTUATION

English employs various symbols to *punctuate* a sentence, to indicate the order in which its phrases are to be interpreted. Thus:

The teacher said he was stupid.

The teacher, said he, was stupid.

Math also employs various devices (primarily parentheses) to specify order of interpretation or, as it is usually called, *order of execution*. It also employs a set of rules for an unparenthesized phrase, including a hierarchy amongst functions. For example, *power* is executed before *times*, which is executed before *addition*.

J uses only parentheses for punctuation, together with the following rules for unparenthesized phrases:

The right argument of a verb is the value of the entire phrase to its right.

Adverbs are applied first. Thus, the phrase `a */ b` is equivalent to `a (*/) b`, not to `a (*/b)`.

For example:

```
   a=.5
   b=.3
   (a*a)+(b*b)
34
   a*a+b*b
70
   a*(a+(b*b))
70
   (a+b)*(a-b)
16
   a (+*-) b
16
```

The last sentence above includes the *isolated* phrase `+*-` which has thus far not been assigned a meaning. It is called a *trident* or *fork*, and is equivalent to the sentence that precedes it.

A fork also has a monadic meaning, as illustrated for the *mean* below:

```
   c=.2 3 4 5 6
   (+/ % #) c
4
   (+/c)%(#c)
4
```

The verb `#` yields the number of items in its argument.

Do the exercises for this lesson.

6: FORKS

As illustrated in the preceding lesson, an isolated sequence of three verbs is called a *fork*; its monadic and dyadic cases are defined by:

```
    g          g
   / \        / \
  f   h      f   h
  |   |     / \ / \
  y   y    x  y x  y
```

```
  (f g h) y is   (f y) g (h y)
x (f g h) y is (x f y) g (x h y)
```

The diagrams at the upper right provide visual images of the fork.

Before reading the notes at the right (and by using the facts that `%:` denotes the *root* function and `]` denotes the *identity* function), try to state in English the significance of each of the following sentences:

```
   a=. 8 7 6 5 4 3
   b=. 4 5 6 7 8 9
   2 %: b                              Square root of b
2 2.23607 2.44949 2.64575 2.82843 3
   3 %: b                              Cube root of b
1.5874 1.70998 1.81712 1.91293 2 2.08008

   (+/ % #) b                          Arithmetic mean or average
6.5
   (# %: */) b                         Geometric mean
6.26521

   (] - (+/ % #)) b                    Centre on mean (two forks)
_2.5 _1.5 _0.5 0.5 1.5 2.5
   (] - +/ % #) b                      Two forks (fewer parentheses)
_2.5 _1.5 _0.5 0.5 1.5 2.5

   a (+ * -) b                         Dyadic case of fork
48 24 0 _24 _48 _72
   (a^2)-(b^2)
48 24 0 _24 _48 _72

   a (< +. =) b                        Less than or equal
0 0 1 1 1 1
   a<b
0 0 0 1 1 1
   a=b
0 0 1 0 0 0
   a (<: = < +. =) b                   A tautology (<: is less than
1 1 1 1 1 1                            or equal)
```

Do the exercises for this lesson.

7: PROGRAMS

A *program* handed out at a musical evening describes the sequence of musical pieces to be performed. As suggested by its roots *gram* and *pro*, a program is something written in advance of the events it prescribes.

Similarly, the fork `+/ % #` of the preceding lesson is a program that prescribes the computation of the mean of its argument when it is applied, as in the sentence `(+/ % #) b`. However, we would not normally call the procedure a program until we assign a name to it, as illustrated below:

```
   mean=. +/ % #

   mean 2 3 4 5 6
4
   (geomean=. # %: */) 2 3 4 5 6
3.72792
```

Since the program `mean` is a new *verb*, we also refer to a sentence such as `mean=. +/ % #` as *verb definition* (or *definition*), and to the resulting verb as a *defined* verb or function.

Defined verbs can be used in the definition of further verbs in a manner sometimes referred to as *structured programming*. For example:

```
   MEAN=. sum % #
    sum=. +/

   MEAN 2 3 4 5 6
4
```

Entry of a verb alone (without an argument) displays its definition. For example:

```
    mean
┌─────┬─┬─┐
│┌─┬─┐│%│#│
││+│/││ │ │
│└─┴─┘│ │ │
└─────┴─┴─┘

    MEAN
┌───┬─┬─┐
│sum│%│#│
└───┴─┴─┘

    sum
┌─┬─┐
│+│/│
└─┴─┘
```

Do the exercises for this lesson.

8: BOND CONJUNCTION

A dyad such as ^ can be used to provide a family of monadic functions. For example:

```
   ]b=. i.7
0 1 2 3 4 5 6
   b^2                              Squares
0 1 4 9 16 25 36
   b^3                              Cubes
0 1 8 27 64 125 216
   b^0.5                            Square roots
0 1 1.41421 1.73205 2 2.23607 2.44949
```

The *bond* conjunction & can be used to bind an argument to a dyad in order to produce a corresponding defined verb. For example:

```
   square=. ^&2                  Square function (power and 2)
   square b
0 1 4 9 16 25 36
   (sqrt=. ^&0.5) b              Square root function
0 1 1.41421 1.73205 2 2.23607 2.44949
```

A left argument can be similarly bound:

```
   Log=. 10&^.                   Base-10 Logarithm
   Log 2 4 6 8 10 100 1000
0.30103 0.60206 0.778151 0.90309 1 2 3
```

Such defined verbs can of course be used in forks. For example:

```
   in29=. 2&< *. <&9             Interval test
   in29 0 1 2 5 8 13 21
0 0 0 1 1 0 0
   IN29=. in29 # ]               Interval selection
   IN29 0 1 2 5 8 13 21
5 8
   LOE=. <+.=
   5 LOE 3 4 5 6 7
0 0 1 1 1
   integertest=. <. = ]                 The monad <. is the
   integertest 0 0.5 1 1.5 2 2.5 3      integer part or floor
1 0 1 0 1 0 1
   int=. integertest
   int (i.13)%3
1 0 0 1 0 0 1 0 0 1 0 0 1
```

Do the exercises for this lesson.

9: ATOP CONJUNCTION

The conjunction `@` applies to two verbs to produce a verb that is equivalent to applying the first *atop* the second. For example:

```
   TriplePowersOf2=. (3&*)@(2&^)
   TriplePowersOf2 0 1 2 3 4
3 6 12 24 48
   CubeOfDiff=. (^&3)@-
   3 4 5 6 CubeOfDiff 6 5 4 3
_27 _1 1 27
```

```
   f=. ^@-
   5 f 3
7.38906
   f 3
0.0497871
```

The first function is applied monadically; the second is applied dyadically if possible.

```
   g=. -@^
   5 g 3
_125
   g 3
_20.0855
```

A conjunction, like an adverb, is executed before verbs. Moreover, the *left* argument of either is the entire verb phrase that precedes it. Consequently, some (but not all) of the parentheses in the foregoing definitions can be omitted. For example:

```
   COD=. ^&3@-
   3 4 5 6 COD 6 5 4 3
_27 _1 1 27
   TPO2=. 3&*@(2&^)
   TPO2 0 1 2 3 4
3 6 12 24 48
   tpo2=. 3&*@2&^
domain error
```

An error because the conjunction `@` is defined only for a *verb* right argument

Do the exercises for this lesson.

10: VOCABULARY

Memorizing lists of words is a tedious and ineffectual way to learn a language, and better techniques should be employed:

A) Conversation with a laconic native speaker, that is, one that allows you to do most of the talking.

B) Reading material of interest in its own right.

C) Learning how to use dictionaries and grammars so as to become independent of teachers.

D) Attempting to *write* on any topic of interest in itself.

E) Paying attention to the *structure* of words so that known words will provide clues to the unknown. For example, *program* (already analyzed) is related to *tele* (far off) *gram*, which is in turn related to *telephone*. Even tiny words may possess informative structure: *atom* means not cuttable, from *a* (not) and *tom* (as in *tome* and *microtome*).

In the case of **J**:

A) The computer provides for precise and general conversation.

B) Texts such as *Tangible Math* [1] and *Arithmetic* [2] use the language in a variety of topics.

C) The appended *Dictionary of J* provides a complete and concise dictionary and grammar.

D) *Programming in J* [3] provides guidance in writing programs, and most any topic provides problems of a wide range of difficulty.

E) Words possess considerable structure, as in `+:` and `-:` and `*:` and `%:` for *double, halve, square,* and *square root*. Moreover, a beginner can assign and use mnemonic names appropriate to any native language, as in `sqrt=.%:` and `entier=.<.` (French) and `sin=.1&o.` and `sind=.1&o.@(%&180@o.)` (for sine in degrees).

We will hereafter introduce and use new primitives with little or no discussion, assuming that the reader will experiment with them on the computer, consult the dictionary to determine their meanings, or perhaps infer their meanings from their structure.

For example, the appearance of the word `o.` suggests a circle; it was used dyadically above to define the sine (one of the *circular* functions), and monadically for the function *pi times*.

Do the exercises for this lesson.

11: HOUSEKEEPING

In an extended session it may be difficult to remember the names assigned to verbs and nouns; the *foreign* conjunction `!:` (detailed in Appendix D of the dictionary) provides facilities for displaying and erasing them. For example:

```
   b=. 3* a=. i. 6
   sum=. +/
   tri=. sum\ a
   names=. 4!:1
   names 2
|a|b|tri|
   names 3
|names|sum|
   erase=. 4!:55
   erase <'tri'
1
   names 2 3
|a|b|erase|names|sum|
   erase names 2
1 1
   names 2 3
|erase|names|sum|
```

It is also useful to be able to *save* a record of a session (that is, a record of all names and their referents) in a specified *locale*. Thus:

```
   copy=. 2!:4
   save=. 2!:2
   save <'abc'
1
   erase names 2 3
1 1 1 1 1
   sum
value error
   2!:4 <'abc'
1
   names 2 3
|copy|erase|names|save|sum|
```

Do the exercises for this lesson.

12: POWER AND INVERSE

The power conjunction `^:` is analogous to the power function `^` . For example:

```
   ]a=.10^ b=. i.5
1 10 100 1000 10000
   b
0 1 2 3 4
   %:a
1 3.16228 10 31.6228 100
   %: %: a
1 1.77828 3.16228 5.62341 10
   %: ^: 2 a
1 1.77828 3.16228 5.62341 10
   %: ^: 3 a
1 1.33352 1.77828 2.37137 3.16228

   %: ^: b a
1      10      100     1000   10000
1 3.16228       10 31.6228      100
1 1.77828 3.16228 5.62341       10
1 1.33352 1.77828 2.37137 3.16228
1 1.15478 1.33352 1.53993 1.77828

   (cos=. 2&o.) ^: b d=.1
1 0.540302 0.857553 0.65429 0.79348
   ] y=. cos ^: _ d
0.739085
   y=cos y
1
```

Successive applications of `cos` appear to be converging to a limiting value; the infinite power (`cos ^: _`) yields this limit.

A right argument of `_1` produces the *inverse* function. Thus:

```
   %: ^: _1 b
0 1 4 9 16

   *: b
0 1 4 9 16

   %: ^: (-b) b
0 1     2       3       4
0 1     4       9      16
0 1    16      81     256
0 1   256    6561   65536
0 1 65536 4.30467e7 4.29497e9
```

Do the exercises for this lesson.

13: READING AND WRITING

Cover the **right** side of the page and make a serious attempt to translate the sentences on the left to English; that is, state succinctly in English what the verb defined by each sentence does. Use any available aids, including the dictionary and experimentation on the computer:

`f1=. <:`	Decrement (monad); Less or equal
`f2=. f1&9`	Less or equal 9
`f3=. f2 *. >:&2`	Interval test 2 to 9 (inclusive)
`f4=. f3 *. <. = ]`	In 2 to 9 and integer
`f5=. f3 +. <. = ]`	In 2 to 9 or integer
`g1=. %&1.8`	Divide by 1.8
`g2=. g1^:_1`	Multiply by 1.8
`g3=. -&32`	Subtract 32
`g4=. g3^:_1`	Add 32
`g5=. g1@g3`	Celsius from Fahrenheit
`g6=. g5^:_1`	Fahrenheit from Celsius
`h1=. >./`	Maximum over list (monad)
`h2=. h1-<./`	Spread. Try `h2 b` with a parabola: `b=. (-&2 * -&3) -:i.12`
`h3=. h1@]-i.@[*h2@]%<:@[`	Grid. Try `10 h3 b`
`h4=. h3 <:/ ]`	Barchart. Try `10 h4 b`
`h5=. {&' *' @ h4`	Barchart. Try `10 h5 b`

After entering the foregoing definitions, enter each verb name alone to display its definition, and learn to interpret the resulting displays.

Cover the **left** side of the page, and translate the English definitions on the right back into **J**.

Do the exercises for this lesson.

14: FORMAT

A numeric table such as:

```
   ]t=.(i.4 5)%3
       0 0.333333 0.666667        1 1.33333
1.66667        2  2.33333 2.66667        3
3.33333  3.66667        4 4.33333 4.66667
       5  5.33333  5.66667        6 6.33333
```

can be rendered more readable by *formatting* it to appear with a specified width for each column, and with a specified number of digits following the decimal point. For example:

```
   ]f=. 6.2 ": t
  0.00  0.33  0.67  1.00  1.33
  1.67  2.00  2.33  2.67  3.00
  3.33  3.67  4.00  4.33  4.67
  5.00  5.33  5.67  6.00  6.33
```

The integer part of the left argument of the format function specifies the column width, and the first digit of the fractional part specifies the number of digits to follow the decimal point.

Although the formatted table *looks* much like the original table `t`, it is a table of *characters*, not of numbers. For example:

```
   $t
4 5
   $f
4 30
   +/t
10 11.3333 12.6667 14 15.3333
   +/f
domain error
```

However, the verb *do* or *execute* (`".`) applied to such a character table yields a corresponding numeric table:

```
   ". f
   0 0.33 0.67    1 1.33
1.67    2 2.33 2.67    3
3.33 3.67    4 4.33 4.67
   5 5.33 5.67    6 6.33

   3* ". f
   0  0.99  2.01     3  3.99
5.01     6  6.99  8.01     9
9.99 11.01    12 12.99 14.01
  15 15.99 17.01    18 18.99
```

Do the exercises for this lesson.

15: PARTITIONS

The function `sum=. +/` applies to an entire list argument; to compute *partial sums* or *subtotals*, it is necessary to apply it to each prefix of the argument. For example:

```
   a=. 1 2 3 4 5 6
   sum=. +/
   sum a
21
   sum\ a
1 3 6 10 15 21
```

The symbol `\` denotes the *prefix* adverb, which applies its argument (in this case `sum`) to each prefix of the eventual argument. The adverb `\.` applies similarly to suffixes:

```
   sum\. a
21 20 18 15 11 6
```

The monad `<` simply *boxes* its arguments, and the verbs `<\` and `<\.` therefore show the effects of partitions with great clarity. For example:

```
   <1 2 3
┌─────┐
│1 2 3│
└─────┘
   (<1),(<1 2),(<1 2 3)
┌─┬───┬─────┐
│1│1 2│1 2 3│
└─┴───┴─────┘
   <\ a
┌─┬───┬─────┬───────┬─────────┬───────────┐
│1│1 2│1 2 3│1 2 3 4│1 2 3 4 5│1 2 3 4 5 6│
└─┴───┴─────┴───────┴─────────┴───────────┘
   <\. a
┌───────────┬─────────┬───────┬─────┬───┬─┐
│1 2 3 4 5 6│2 3 4 5 6│3 4 5 6│4 5 6│5 6│6│
└───────────┴─────────┴───────┴─────┴───┴─┘
```

The *oblique* adverb `/.` partitions a *table* along diagonal lines. Thus

```
   ]t=.1 2 1 */ 1 3 3 1
1 3 3 1
2 6 6 2
1 3 3 1
   </. t
┌─┬───┬─────┬─────┬───┬─┐
│1│3 2│3 6 1│1 6 3│2 3│1│
└─┴───┴─────┴─────┴───┴─┘
   sum/. t
1 5 10 10 5 1
   10 #. sum/. t
161051
   121*1331
161051
```

Do the exercises for this lesson.

16: DEFINED ADVERBS

Names may be assigned to adverbs, as they are to nouns and verbs:

```
   a=.1 2 3 4 5
   prefix=. \
   < prefix 'abcdefg'
```

a	ab	abc	abcd	abcde	abcdef	abcdefg

Moreover, new adverbs result from a string of adverbs (such as /\), and from a conjunction together with one of its arguments. Such adverbs can be *defined* by assigning names. For example:

```
   InsertPrefix=. /\
   + InsertPrefix a
1 3 6 10 15
   with3=. &3
   % with3 a
0.333333 0.666667 1 1.33333 1.66667
   ^ with3 a
1 8 27 64 125
   inverse=. ^: _1
   *: inverse a
1 1.41421 1.73205 2 2.23607
   ten=. 10&
   ^. ten 5 10 20 100
0.69897 1 1.30103 2
   #. ten 3 6 5
365
```

The adverb ~ *commutes* or *crosses* the connections to its argument verb, as illustrated below:

```
   3-5
_2
   3-~5                         Three from five
2
   3%~5                         Three into five
1.66667
```

The monad `f~` replicates its argument to provide both arguments to the dyad `f` . For example:

```
   +/~i.3
0 1 2
1 2 3
2 3 4
```

Do the exercises for this lesson.

17: WORD FORMATION

The interpretation of a written English sentence begins with word formation. The basic process is based on spaces to separate the sentence into units, but is complicated by matters such as apostrophes and punctuation marks: *'twas* and *Brown's* and *Ross'* are each single units, but *however,* is not (since the comma is a separate unit).

The following lists of characters represent sentences in **J**, and can be executed by applying the *do* or *execute* function `".` :

```
   m=. '3 %: y.'
   d=. 'x.%: y.'
   x.=. 4
   y.=. 27 4096
   ". m
3 16
   do=. ".
   do d
2.27951 8
```

The word formation rules of **J** are prescribed in Section I of the dictionary. Moreover, the word-formation function `;:` can be applied to the string representing a sentence to produce a boxed list of its words:

```
   ;: m
┌─┬──┬──┐
│3│%:│y.│
└─┴──┴──┘
```

```
   words=. ;:
   words d
┌──┬──┬──┐
│x.│%:│y.│
└──┴──┴──┘
```

The rhematic rules of **J** apply reasonably well to English phrases:

```
  words p=. 'Nobly, nobly, Cape St. Vincent'
┌─────┬─┬─────┬─┬────┬───┬───────┐
│Nobly│,│nobly│,│Cape│St.│Vincent│
└─────┴─┴─────┴─┴────┴───┴───────┘

   >words p
Nobly
,
nobly
,
Cape
St.
Vincent
```

Do the exercises for this lesson.

18: NAMES

In addition to the normal names used thus far, there are four further classes:

1) `$:` is used for self-reference, allowing a verb to be defined recursively without necessarily assigning a name to it. Its use is discussed in Lesson 23.

2) The names `x.` and `y.` and `$.` are used in explicit definition, discussed in Lesson 19. The first two denote the arguments used in explicit definition, and the last denotes the list (or *suite*) that controls the sequence of execution of the set of sentences specified in the definition.

3) A name ending with a colon is a *given* name whose referent once assigned cannot be changed. For example:

```
   months:=. 31 28 31 30 31 30 31 31 30 31 30 31
   +/months:
365
   months:=. months: + 1=i.12
not reassignable
```

4) A name that includes an underbar (_) is a *locative*. Names used in a *locale* `F` can be referred to in another locale `G` by using the prefix `F` in a locative name of the form `F_pqr`, thus avoiding conflict with otherwise identical names in the locale `G`.

The referent of a locative can be established in either of two ways:

a) By assignment, as in `F_pqr=. i. 5.`

b) By *saving* a session in the manner discussed in Lesson 11; the names established in the session can thereafter be referred to by using the locale as a prefix in a locative name. For example:

```
   names=. 4!:1
   copy=. 2!:4
   save=. 2!:2
   save <'TOOLS'
1
   4!:55 names 3
1 1 1
   TOOLS_copy <'TOOLS'
1
   names 3
+----+-----+----+
|copy|names|save|
+----+-----+----+
```

Do the exercises for this lesson.

19: EXPLICIT DEFINITION

The character lists:

```
   m=. '3 %: y.'
   d=. 'x.%: y.'
```

that were analyzed and executed as sentences in Lesson 17, can be used with the *explicit definition* conjunction : to produce a verb:

```
   roots=. m : d
   roots 27 4096
3 16
   4 roots 27 4096
2.27951 8
```

The space before the colon is essential because it would otherwise combine with the m to produce the word m: as illustrated by using the word formation function:

```
   words=. ;:
   words 'm:d'                       words 'm :d'
┌──┬─┐                          ┌─┬─┬─┐
│m:│d│                          │m│:│d│
└──┴─┘                          └─┴─┴─┘
```

The arguments of the conjunction : may also be tables or boxed lists representing a number of sentences. These sentences are selected in an order determined by the *suite* $. which is initially set to i.n, where n is the number of sentences. Execution terminates when the elements of the suite are exhausted, and the result of the function defined is the result of the last sentence executed.

Except for the fact that it is local to the function being defined, $. behaves like an ordinary noun, and can be re-assigned a new value at any point:

```
   b=. 'r=. y.';'$.=. x.#2';'r=. r,+/_2{.r'
   fib=. '' : b
   6 fib 1 1
1 1 2 3 5 8 13 21
   fib
┌┬─┬──────────────┐
││:│r=. y.        │
││ │$.=. x.#2     │
││ │r=. r,+/_2{.r │
└┴─┴──────────────┘
```

Explicit definition of adverbs and conjunctions occurs in exercises.

Do the exercises for this lesson.

20: TACIT EQUIVALENTS

Verbs may be defined either explicitly or tacitly. In the case of a one-sentence explicit definition, of either a monadic or dyadic case, the corresponding tacit definition may be obtained by using the adverb `:20` as illustrated below:

```
   s=. '(+/y.) % (#y.)'
   mean=. s : ''
   mean
+--------------+-+-+
|(+/y.) % (#y.)|:| |
+--------------+-+-+
   (mean = MEAN) ?20#100
1
```

```
   MEAN=. s : 20
   MEAN
+----------------+-+-----------+
|+-------+-+-+   |%|+-+-+-+    |
||+-+-+  |@|]|   | ||#|@|]|    |
|||+|/|  | | |   | |+-+-+-+    |
||+-+-+  | | |   | |           |
|+-------+-+-+   | |           |
+----------------+-+-----------+
```

The tacit definition produced by `:20` is not necessarily the briefest possible. For example, enter `m=. +/%#` and use and display `m`.

The explicit form of definition is likely to be more familiar to computer programmers than the tacit form. Translations provided by the adverb `:22` may therefore be helpful in learning tacit programming.

An explicit definition of a conjunction may be translated similarly by the adverb `:12`. For example:

```
   s=. 'y.f.^:_1 @ (x.f.)&(y.f.)'
   under=. s : 2
   times=. + under ^.
   3 times 4
12
   3 + (u=. s : 22) ^. 4
12

   u
+-----------------------------------------+-+--+
|+-------------------------+-+----------+ |&|f.|
||+--+-------------------+ |@|+--+--+  | | |  |
|||].|+--+-----------+   | | ||[.|f.|  | | |  |
|||  ||f.|+--+--+    |   | | |+--+--+  | | |  |
|||  ||  ||^:|_1|    |   | | |          | | |  |
|||  ||  |+--+--+    |   | | |          | | |  |
|||  |+--+-----------+   | | |          | | |  |
||+--+-------------------+ | |          | | |  |
|+-------------------------+-+----------+ | |  |
+-----------------------------------------+-+--+

   3 + ( UNDER=. (].(^:_1))@([.&].) ) ^. 4
12
```

Do the exercises for this lesson.

21: RANK

The shape (`$`), tally (`#`), and rank (`#$`), of a noun are illustrated by the noun `report`, which may be construed as a report covering two years of four quarters of three months each:

```
   ]report=. i. 2 4 3
 0  1  2
 3  4  5
 6  7  8
 9 10 11

12 13 14
15 16 17
18 19 20
21 22 23
```

```
   $report
2 4 3
   #report
2
   #$report
3
```

The last *k* axes determine a *k-cell* of a noun; the 0-cells of `report` are the atoms (such as `4` and `14`), the 1-cells are the three-element quarterly reports, and the two-cells (or *major cells* or *items*) are the two four-by-three yearly reports.

The rank conjunction `"` is used in the phrase `f"k` to apply a function `f` to each of the `k`-cells of its argument. For example:

```
   ,report
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
   ,"2 report
 0  1  2  3  4  5  6  7  8  9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
   <@i. s=. 2 5
┌─────────┐
│0 1 2 3 4│
│5 6 7 8 9│
└─────────┘
   <@i."0 s
┌───┬─────────┐
│0 1│0 1 2 3 4│
└───┴─────────┘
```

Both the left and right ranks of a dyad may be specified. For example:

```
   i,"0 1 i=.0 1 2
0 0 1 2
1 0 1 2
2 0 1 2
```

```
   i,"1 1 i
0 1 2 0 1 2
   ,"1~i
0 1 2 0 1 2
```

Do the exercises for this lesson.

22: GERUND AND AGENDA

In English, a *gerund* is a noun that carries the force of a verb, as does the noun *cooking* in the phrase *the art of cooking*; and *agenda* is a list of items for action.

The *tie* conjunction ` applies to two verbs to form a gerund, from which elements can be chosen for execution. For example, if the agenda conjunction `@.` is applied to a gerund, its verb right argument provides the results that choose the elements. For example:

```
   g=. +`^
   a=.<
   2 a 3
1
   2 g@.a 3
8
   3 g@.a 2
5
   +:`-:`*:`%: @. (4&|@<.)"0 i. 10
0 0.5 4 1.73205 8 2.5 36 2.64575 16 4.5
```

The verb produced by `g@.a` is often called a *case* or *case statement*, since it selects one of the "cases" of the gerund for execution.

The *insert* adverb `/` applies to a gerund in a manner analogous to its application to a verb. For example:

```
   c=.3 [ x=. 4 [ power=. _1
   g/ c,x,power
3.25
   3+x^_1
3.25
```

The elements of the gerund are repeated as required. For example:

```
   +`*/1,x,3,x,3,x,1
125
```

The last sentence above corresponds to Horner's efficient evaluation of the polynomial with coefficients `1 3 3 1` and argument `x`.

Do the exercises for this lesson.

23: RECURSION

The factorial function `!` is commonly defined by the statement that factorial of `n` is `n` times factorial of `n-1`, and by the further statement that factorial of `0` is `1`. Such a definition is called *recursive*, because the function being defined recurs in its definition.

A case statement can be used to make a recursive definition, the case that employs the function under definition being chosen repeatedly until the terminating case is encountered. For example:

```
   factorial=. 1:`(]*factorial@<:) @. *
   factorial "0 i.6
1 1 2 6 24 120
```

In the sentence `(sum=. +/) i.5` the verb defined by the phrase `+/` is assigned a name before being used, but in the sentence `+/ i.5` it is used anonymously.

In the definition of `factorial` above, it was essential to assign a name to make it possible to refer to it within the definition. However, the word `$:` provides *self-reference* that permits anonymous recursive definition. For example:

```
   1:`(]*$:@<:) @. * "0 i. 6
1 1 2 6 24 120
```

Do the exercises for this lesson.

24: ITERATION

The repetition of a process, or of a sequence of similar processes, is called *iteration*. Much iteration is implicit, as in `*/b` and `a*/b`, and `a*b`; explicit iteration is provided by the power conjunction `^:` :

```
   (cos=. 2&o.) ^: (i.6) b=. 1
1 0.540302 0.857553 0.65429 0.79348 0.701369
   ]y=. cos^:_ b                          y=cos y
0.739085                               1
```

The example `cos^:_` illustrates the fact that infinite iteration is meaningful (i.e., terminates) if the process applied converges to a limit.

Controlled iteration of a process `p` is provided by `p^:q`, where the result of `q` determines the number of applications of `p` performed before again applying `q`. A zero result from `q` terminates the process.

For example, to add to a beginning value `3` the sum of successive negative powers of `4`, beginning with `_1`, and continuing as long as the ratio of the sum to the next power exceeds `1000`:

```
   f=. +`^/ , 1&{ , <:@{:
   g=. 1000&>@(%`^/)
   f 3,4,_1                       {.@(f^:g) 3 4 _1
3.25 4 _2                       3.33203
```

If `f` is a continuous function, and if `f i` and `f j` differ in sign, then there must be a *root* `r` between `i` and `j` such that `f r` is zero; the list `b=.i,j` is said to *bracket* a root. A narrower bracket is provided by the mean of `b` together with that element of `b` whose result differs in sign from its result . Thus:

```
   (f=. 4&-@%:) 16,s=. 1 34
0 3 _1.83095
   m=. +/%#                       sos=.m ~:&(*@f) ]
   f m s                          sos s   Select opposite sign
_0.1833                          1 0
   (br=. m, sos # ])^:0 1 2 3 s
     1   34
   17.5    1
   9.25 17.5
 13.375 17.5
   br^:_ s
16 16 16
   BR=. m, (=/ < ])@sos # ]          Select none if signs are
   BR^:_ s                           equal (i.e., converged)
16              Do the exercises for this lesson.
```

25: TRAINS

The train of nouns in the English phrase *Ontario museum Egyptian collection* represents a single noun. Similarly, the fork and hook discussed in Lesson 6 and its exercises permit the use of arbitrarily long trains of verbs to produce a verb.

Lesson 16 introduced the use of trains of adverbs, and of conjunctions and nouns or verbs, to represent adverbs. *Conjunctions* may also be produced by trains of adverbs and conjunctions in a manner analogous to hooks and forks.

For example, the case diagrammed on the right below can be used as follows:

```
   cj=.\@\
   < cj (+/) a=.i.3 3
┌─────┬─────┬─────────┐
│0 1 2│0 1 2│0  1  2  │
│     │3 5 7│3  5  7  │
│     │     │9 12 15  │
└─────┴─────┴─────────┘
   (<\)@(+/\) a
┌─────┬─────┬─────────┐
│0 1 2│0 1 2│0  1  2  │
│     │3 5 7│3  5  7  │
│     │     │9 12 15  │
└─────┴─────┴─────────┘
```

```
    c
   / \
 a1   a2
 |     |
 x     y
```

```
   (*/) cj (+/) a
0  1   2
0  5  14
0 60 210
```

The explicit form of defining conjunctions treated in the exercises of Lesson 19 can be used to produce an equivalent conjunction `CJ` as shown below. The corresponding tacit definition produced by `s : 22` can be simplified to the form used in defining `cj` above:

```
   s=.'(x.f.\)@(y.f.\)'
   CJ=. s : 2
   (*/) CJ (+/) a
0  1   2
0  5  14
0 60 210
   s : 22
┌───────┬─┬───────┐
│┌──┬─┐│@│┌──┬─┐│
││f.│\││ ││f.│\││
│└──┴─┘│ │└──┴─┘│
└───────┴─┴───────┘
```

Do the exercises for this lesson.

26: PERMUTATIONS

Anagrams are familiar examples of the important notion of *permutations*:

```
   w=. 'STOP'
   3 2 0 1 { w
POST
   2 3 1 0 { w
OPTS
   3 0 2 1 { w
PSOT
```

The left arguments of `{` above are themselves permutations of the list `i.4`, and are examples of *permutation vectors*, used to represent permutation functions in the form `p&{` .

If `p` is a permutation vector, the phrase `p&C.` also represents the permutation `p&{` . However, other cases of the *cycle* function `C.` are distinct from the *from* function `{` . In particular, `C. p` yields the *cycle* representation of the permutation `p`. For example:

```
   ]c=. C. p=. 2 4 0 1 3
┌───┬─────┐
│2 0│4 3 1│
└───┴─────┘
   c C. 'ABCDE'
CEABD
   C. c
2 4 0 1 3
```

Each of the boxed elements of a cycle specify a list of positions that cycle among themselves; in the example above, the element from position `3` moves to position `4`, element `1` moves to `3`, and element `4` to `1`.

If all `!n` permutations of order `n` are listed in a table in increasing order (when considered as base-`n` numbers), they can be identified by their row indices `i.!n`. This index is the *atomic* representation of the permutation; the corresponding permutation is effected by the function `A.`:

```
   1 A. 'ABCDE'
ABCED
   A. 0 1 2 4 3
1
   (i.!3) A. i.3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
   (i.!3) A. 'ABC'
ABC
ACB
BAC
BCA
CAB
CBA
```

Do the exercises for this lesson.

27: LINEAR FUNCTIONS

A function `f` is said to be *linear* if `f(x+y)` equals `(f x)+(f y)` for all arguments `x` and `y`. For example:

```
   f=.3&|.@+:@|.
   ]x=. i.# y=.2 3 5 7 11
0 1 2 3 4
   x+y                                  f x+y
2 4 7 10 15                          8 4 30 20 14
   (f x),:(f y)                         (f x)+(f y)
2 0  8  6  4                         8 4 30 20 14
6 4 22 14 10
```

A linear function can be defined equivalently as follows: `f` is linear if `f@:+` and `+&f` are equivalent. For example:

```
   x f@:+ y
4 8 14 20 30
   x +&f y
4 8 14 20 30
```

If `f` is a linear function, then `f y` can be expressed as the *matrix product* `mp&M y`, where

```
   mp=. +/ . *
   M=. f I=. =/~i.#y                 I is an identity matrix
   mp&M y                               f y
6 4 22 14 10                         6 4 22 14 10
```

Conversely, if `m` is any square matrix of order `#y`, then `m&mp` is a linear function on `y`, and if `m` is invertible, then `(%.m)&mp` is its inverse:

```
   x=.1 2 3 [ y=. 2 3 5
   ]m=. ? 3 3$9
5 7 3
7 2 3
4 4 2
   ]n=. %.m
 _1.33333 _0.333333  2.5
_0.333333 _0.333333    1
  3.33333   1.33333 _6.5

   g=. mp&m
   h=. mp&n

   x g@:+ y                   x +&g y
82 63 40                   82 63 40
   g h y
2 3 5
```

Do the exercises for this lesson.

28: OBVERSE AND UNDER

The result of `f^:_1` is called the *obverse* of the function `f`; if `f=. g :. h`, this obverse is `h`, and it is otherwise an inverse of `f` . Inverses are provided for over 25 primitives (including the case of the square root illustrated in Lesson 12), as well as invertible monads such as `-&3` and `10&^.` and `2&o.` . Moreover, `u@v^:_1` is given by `(v^:_1)@(u^:_1)`. For example:

```
   fFc=. (32&+)@(*&1.8)
   ]b=.fFc _40 0 100
_40 32 212
   cFf=. fFc^:_1
   cFf b
_40 0 100
```

The result of the phrase `f &. g` is the verb `(g^:_1) @ (f & g)` . The function `g` can be viewed as *preparation* (which is done before and undone after) for the application of the "main" function `f` . For example:

```
   b=. 0 0 1 0 1 0 1 1 0 0 0
   sup=. </\                       Suppress ones after the first
   sup b
0 0 1 0 0 0 0 0 0 0 0
   |. sup |. b                     Suppress ones before the last
0 0 0 0 0 0 0 1 0 0 0
   sup&.|. b                       "
0 0 0 0 0 0 0 1 0 0 0
   3 +&.^. 4                       Multiply by applying the
12                                 exponential to the sum of
   (^.3)+(^.4)                     logarithms
2.48491
   ^ (^.3)+(^.4)
12
   ]c=. 1 2 3;4 5;6 7 8
+-----+---+-----+
|1 2 3|4 5|6 7 8|
+-----+---+-----+
   |.&.> c                         Open, reverse, and then box
+-----+---+-----+
|3 2 1|5 4|8 7 6|
+-----+---+-----+
```

Do the exercises for this lesson.

29: IDENTITY FUNCTIONS

The monads `0&+` and `%&1` are *identity* functions, and `0` and `1` are said to be *identity elements* of the dyads `+` and `%` respectively. Insertion on an empty list yields the identity element of the dyad inserted. For example:

```
   +/ i.0                    +/''              +/0{. 2 3 5
0                         0                 0
   */i.0                     */''              */0{. 2 3 5
1                         1                 1
```

These results are useful in partitioning lists; they ensure that certain obvious relations continue to hold even when one of the partitions is empty. For example:

```
   +/ a=. 2 3 5 7 11
28
   (+/4{.a)+(+/4}.a)
28
   (+/0{.a)+(+/0}.a)
28
   */a
2310
   (*/4{.a)*(*/4}.a)
2310
   (*/0{.a)*(*/0}.a)
2310
```

Do the exercises for this lesson.

30: EXPERIMENTS

Although this introduction is not exhaustive, it should prepare the reader to understand and use the appended dictionary of **J**, which is. The following examples suggest experiments that might prove interesting to pursue in the dictionary:

VERBS

```
   {'ab';'+-*';'cd'
```

a+c	a+d
a-c	a-d
a*c	a*d

b+c	b+d
b-c	b-d
b*c	b*d

```
   ~. 'mississippi'
misp
   = 'mississippi'
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 0 0 1 0 0 1
0 0 1 1 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0

   j./i. 2 3 4
0j12 1j13  2j14  3j15
4j16 5j17  6j18  7j19
8j20 9j21 10j22 11j23
```

ADVERBS

```
   2 <\ i. 5
```

0 1	1 2	2 3	3 4

```
   2 <\. i. 5
```

2 3 4	0 3 4	0 1 4	0 1 2

```
   1 2 3 1 3 2 1+//.i.7
9 6 6

   'AB' 2 4} 'abcdef'
abAdBf
```

CONJUNCTIONS

```
   <;.(_1)0 3 5 7 0 2 0 6 8
```

3 5 7	2 6 8

```
   3(= , =!.0) *: %: 3
1 0

   3 |.!.0 i.8
3 4 5 6 7 0 0 0
```

Further material may be found in References [4-6]

Do the exercises for this lesson.

31: SAMPLE TOPICS

These 56 frames provide an informal introduction to J, designed to be used in conjunction with the dictionary and at the keyboard of a J system. They are also designed to be used *inductively*, as follows:

* Read one or two sentences and their results (which begin at the left margin), and attempt to state clearly in English what each sentence does.

* Enter similar sentences to test the validity of your statements.

* Consult the dictionary to confirm your understanding of the meaning of primitives such as `i.` (used both with one argument and with two). Use the vocabulary of Appendix E as an index to pages in the dictionary.

* Enter *parts* of a complex sentence, such as `i. 28` and `j+/i.28` in the case of `(j+/i.28){a.`.

SPELLING

```
  phrase=.'index=.a.i.''aA'''
  ;:phrase
```

index	=.	a.	i.	'aA'

```
   $ ;:phrase
5
   >;:phrase
index
=.
a.
i.
'aA'
   do=. ".
   do phrase
97 65
   do 'abc =. 3 1 4 2'
3 1 4 2
   abc
3 1 4 2
```

ALPHABET and NUMBERS 31

```
   $ a.
256
   j=. a. i. 'aA'
   j
97 65
   j +/ i. 8
97 98 99 100 101 102 103 104
65 66 67  68  69  70  71  72
   (j+/i.28){a.
abcdefghijklmnopqrstuvwxyz{|
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
   a.{~j+/i.28
abcdefghijklmnopqrstuvwxyz{|
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
   i. 2 5
0 1 2 3 4
5 6 7 8 9
   */~0j1 _1 0j_1 1
  _1 0j_1    1  0j1
0j_1    1  0j1   _1
   1  0j1   _1 0j_1
 0j1   _1 0j_1    1
```

GRAMMAR

```
   fahrenheit =. 50
   (fahrenheit - 32) * 5 % 9
10
   prices =. 3 1 4 2
   orders =. 2 0 2 1
   orders * prices
6 0 8 2
   +/ orders * prices
16
   +/ \ 1 2 3 4 5
1 3 6 10 15
   2 3 * / 1 2 3 4 5
2 4 6  8 10
3 6 9 12 15
   cube=. ^&3
   cube i. 9
0 1 8 27 64 125 216 343 512
```

PARTS OF SPEECH

`50 fahrenheit`	Nouns/Pronouns
`+ - * % cube`	Verbs/Proverbs
`/ \`	Adverbs
`&`	Conjunction
`=.`	Copula
`( )`	Punctuation

FUNCTION TABLES Just as the behaviour of *addition* is made clear by *addition tables* in elementary school, so the behaviour of other verbs (or *functions*) can be made clear by function tables.

The next few frames show how to make function tables, and how to use the *utility* functions `over` and `by` to border them with their arguments to make them easier to interpret.

Study the tables shown, and make tables for other functions (such as < <. %) suggested by the summary table.

Utility functions such as `over` and `by` are meant for *use* rather than for immediate study, but the *tacit* programming used to define them will be used throughout, with *explicit* programming (more familiar in other programming languages) treated on page 40. You may skip ahead to it, or to tacit programming on page 36.

FUNCTION TABLES

```
   prices=. 3 1 4 2
   orders=. 2 0 2 1
   prices * orders
6 0 8 2
   table=. prices */ orders
   table
6 0 6 3
2 0 2 1
8 0 8 4
4 0 4 2
```

TO BORDER A TABLE BY ARGUMENTS:

```
   over=.({.;}.)@":@,
   by=.' '&;@,.@[,.]

prices by orders over table
┌─┬───────┐
│ │2 0 2 1│
├─┼───────┤
│3│6 0 6 3│
│1│2 0 2 1│
│4│8 0 8 4│
│2│4 0 4 2│
└─┴───────┘
```

TABLES

```
   n=. 0 1 2 3
   n +/ n        Addition table
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
   */ ~ n        Times table
0 0 0 0
0 1 2 3
0 2 4 6
0 3 6 9
   ^/ ~ i. 4     Power table
1 0 0  0
1 1 1  1
1 2 4  8
1 3 9 27
   +./~ 0 1      Or table
0 1
1 1
```

TABLES (Letter Frequency)

```
   text=. ' i sing of olaf '
   text=.text,'glad and big'

   alph=. ' abcdefghijklmno'
   alph=. alph,'pqrstuvwxyz'

   '01'{~10{.alph=/text
1010000100100001000001000100 0
0000000000000010000100100000 0
0000000000000000000000000010 0
0000000000000000000000000000 0
0000000000000000000010001000 0
0000000000000000000000000000 0
0000000001000010000000000000 0
0000001000000000010000000000 1
0000000000000000000000000000 0
0100100000000000000000000001 0

   ]LF=. 2 13$+/"1 alph=/text
7 3 1 0 2 0 2 3 0 3 0 0 2
0 2 2 0 0 0 1 0 0 0 0 0 0
```

TABLES

```
   |/ ~ 1+i. 5
0 0 0 0 0
1 0 1 0 1
1 2 0 1 2
1 2 3 0 1
1 2 3 4 0
   +/ 0= |/ ~ j=. 1+i. 15
1 2 2 3 2 4 2 4 3 4 2 6 2 4 4
   2=+/0=|/ ~ j
0 1 1 0 1 0 1 0 0 0 1 0 1 0 0
   (2=+/0=|/~ j) # j
2 3 5 7 11 13
   =/~i. 4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
   <:/~i. 4
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1
```

CLASSIFICATION

Classification is a familiar notion. For example, the classification of letters of the alphabet as *vowel, consonant, sibilant,* or *plosive*; the classification of colours as *primary* and *secondary*; and of numbers as *odd*, *even*, *prime*, and *complex*.

It is also very important; it provides the basis for many significant notions, such as graphs, barcharts, and sets.

A classification may be *complete*, (each object falls into at least one class), and it may be *disjoint*, (each object falls into at most one class). A *graph* is a *disjoint* classification corresponding to the non-disjoint classification used to produce a barchart.

The sentence `</\0 0 0 1 0 1 1 0 1` appearing in the bottom right frame on this page illustrates how the phrase `</\` produces a disjoint classification by suppressing all `1`'s after the first.

CLASSIFICATION

```
   x=. 1 2 3 4 5 6 7
   y=. (x-3) * (x-5)
   y
8 3 0 _1 0 3 8
   r=.m-i. 1+(m=. >./y)-<./y
   ]range=. r
8 7 6 5 4 3 2 1 0 _1
   bc=. range <:/ y
   bc
1 0 0 0 0 0 1
1 0 0 0 0 0 1
1 0 0 0 0 0 1
1 0 0 0 0 0 1
1 0 0 0 0 0 1
1 1 0 0 0 1 1
1 1 0 0 0 1 1
1 1 0 0 0 1 1
1 1 1 0 1 1 1
1 1 1 1 1 1 1
```

CLASSIFICATION (Bar Chart)

```
   x=. 1 2 3 4 5 6 7
   y=. (x-3) * (x-5)
   y
8 3 0 _1 0 3 8
range=. m-i.>:(m=. >./y)-<./y
   range
8 7 6 5 4 3 2 1 0 _1
   bc=. range <:/ y
   bc { ' *'
*     *
*     *
*     *
*     *
*     *
**   **
**   **
**   **
*** ***
*******
   </\ 0 0 0 1 0 1 1 0 1
0 0 0 1 0 0 0 0 0
```

CLASSIFICATION (Graphs)

```
   </\bc
1 0 0 0 0 0 1
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 1 0 0 0 1 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 1 0 1 0 0
0 0 0 1 0 0 0
   ' *' {~ </\bc
*     *




 *   *



  * *
   *
```

CLASSIFICATION (Subsets, Key)

```
   2 3 5 +/ . * cct
0 5 3 8 2 7 5 10
   2 3 5 */ . ^ cct
1 5 3 15 2 10 6 30
   2 3 5 >./ . * cct
0 5 3 5 2 5 3 5
   +/cct
0 1 1 2 1 2 2 3
   ]c2=. (2=+/cct)#"1 cct
0 1 1
1 0 1
1 1 0
   2 3 5 >./ . * c2
5 5 3
   2 #. |: cct
0 1 2 3 4 5 6 7
   2 3 5 */ . ^ c2
15 10 6
accounts=.3 1 3  2  6  3 2
credits=.9 7 25 14 31 16 8
   accounts </. credits
┌─────────┬─┬────┬──┐
│9 25 16  │7│14 8│31│
└─────────┴─┴────┴──┘
   accounts +//. credits
50 7 22 31
```

CLASSIFICATION (Dot Prod)

```
   ]m=. 2 3 5,:4 2 1
2 3 5
4 2 1
   ]cct=. |:#: i. 2^3
0 0 0 0 1 1 1 1     Complete
0 0 1 1 0 0 1 1     classification
0 1 0 1 0 1 0 1     table
   m +/ . * cct
0 5 3 8 2 7 5 10
0 1 2 3 4 5 6  7
```

The pattern of the matrix product (`+/ . *`) is illustrated below:

```
     2 3 5 | 0 5 3 8 2 7 5 10
     4 2 1 | 0 1 2 3 4 5 6 7
     ------|----------------
           | 0 0 0 0 1 1 1 1
           | 0 0 1 1 0 0 1 1
           | 0 1 0 1 0 1 0 1
   row0=.2 3 5 [ col3=.0 1 1
   row0 * col3
0 3 5
   +/ row0 * col3
8
   row0 +/ . * cct
0 5 3 8 2 7 5 10
```

SORTING

```
t=.'i sing of olaf glad and big'
   [ tt=. > ;: t
i
sing
of
olaf
glad
and
big
   /: tt
5 6 4 0 2 3 1
   tt /: tt
and
big
glad
i
of
olaf
sing
   (/:~tt) -: (tt /: tt)
1
```

STRUCTURES (Box)

```
   text
i sing of olaf glad and big
   |. text
gib dna dalg falo fo gnis i
   < 'glad'
```

glad

```
u=. (<'glad'),(<'and'),<'big'
   u
```

glad	and	big

```
   |. u
```

big	and	glad

```
   # u
3
   'glad';'and';'big'
```

glad	and	big

PARTITIONS

```
 +/\a=.2 3 5 7 11[b=.'abcdef'
2 5 10 17 28
   <\b
```

a	ab	abc	abcd	abcde	a	etc.

```
   2 <\b
```

ab	bc	cd	de	ef

```
   2 -~/\a
1 2 2 4
   2<\.b
```

cdef	adef	abef	abcf	etc.

```
   </. t=.1 2 1 */ 1 3 3 1
```

1	3 2	3 6 1	1 6 3	2 3	etc.

```
   +//.t
1 5 10 10 5 1
```

STRUCTURES (Open)


```
t=.'i sing of olaf glad and big'
   ]words=. ;:t
```

i	sing	of	olaf	glad	an	etc.

```
   tt=. > words
   tt
i
sing
of
olaf
glad
and
big
   $ tt
7 4

   ;words
isingofolafgladandbig
```

Although the preceding frames have presented rather complex results, they have shown only one example (`cube` of frame 4 of page 1) of *programming* in the sense of assigning a name to a procedure for later use.

Further frames will present many programs. To anyone familiar only with conventional languages and not with *tacit* or *functional* programming, they will not look like programs at all. Nevertheless, tacit programming offers significant advantages: it is brief and analytic, it encourages structured programming, and it is "compiled" in the sense that a program is not re-parsed on execution.

We introduce three *compositions* that facilitate tacit programming: the conjunction `&` that *bonds* a verb to one of its arguments; the conjunction `@` that applies one verb *atop* another; and the *fork* that forms a verb from an isolated list of three verbs. The *hook* is also introduced as a special case of the fork. To display a verb `f` enter `f` alone.

TACIT PROGRAMMING

```
   cubesum=. ^&3@+
   6 cubesum 4
1000
   ^&3 (6+4)
1000
   6 + * - 4
5
   6 (+ * -) 4
20
   (6+4)*(6-4)
20
   mean=. +/ % #
   mean a=.3 4 5
4
   (+/a)%(#a)
4

   centre=. ] - mean
   variance=. mean@*:@centre
   variance i. 100
833.25
```

```
x(f g h)y
    g
   / \
  f   h
 /\   /\
x  y x  y

 (f g h)y
    g
   / \
  f   h
  |   |
  y   y
```

GEOMETRY (2-space)

```
   length=. %:@(+/@*:)
   length 5 12
13
   ]t=. 3 4 7 ,: 0 0 4
3 4 7
0 0 4
   1 |."1 triangle=. t
4 7 3
0 4 0
   ]lsides=.length t-1|."1 t
1 5 5.65685
   ] semiper=. -: +/lsides
5.82843
   area=.%:*/semiper-0,lsides
   area
2
   triangle, %!2
  3   4   7
  0   0   4
0.5 0.5 0.5
   -/ . * t, %!2
2
   -/ . * 1 0 2 {"1 t, %!2
_2
```

Heron's formula

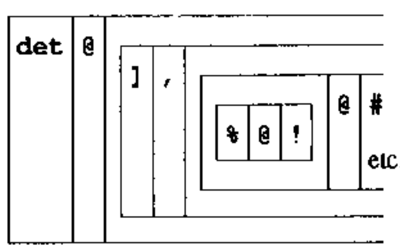
Determinant gives signed area

SYMBOLICS (Insertion, Scan)

```
   minus=.[ , '-'&,@]
   'a' minus 'b'
   |. 'a' minus 'b'
b-a
   minus / list=. 'defg'
d-e-f-g
   minus /\ list
d
d-e
d-e-f
d-e-f-g
   'defg'=. 4 3 2 1
   ". minus / list
2
   ". minus /\ list
4 1 3 2
   times=. [ , '*'&,@]
   list times"0 |. list
d*g
e*f
f*e
g*d
```

GEOMETRY (3-space)

```
]tetrahedron=.1 6 11 e.~i.3 4
0 1 0 0
0 0 1 0
0 0 0 1
   det=. -/ . *
   volume=. det @(] , %@!@#)
   volume tetrahedron
_0.166667
tet=.6 0 3 0,3 6 5 8,:7 4 0 5
   tet
6 0 3 0
3 6 5 8
7 4 0 5
   volume tet
11.5
   volume
```



HOOK (g h) is same as ([g h@])

```
   a=.5 6 7 8
   b=.1 2 3 4
   (*>:) b
2 6 12 20
   a (*>:) b
10 18 28 40
   a (*>:)"0/ b
10 15 7
 25
12 18 24 30
14 21 28 35
16 24 32 40
```

Continued fractions:

```
   (+%) / 1 2 2 2 2 2 2
1.4142
   (+%)/\ 1 2 2 2 2
1 1.5 1.4 1.41667 1.41379
   (+%)/\ 3 7 15
3 3.14286 3.14151
   (+%)/\1 1 1 1 1 1
1 2 1.5 1.66667 1.6 1.625
   (-%)/\ 1 2 2 2 2
1 0.5 0.333333 0.25 0.2
   *~ (+%)/ 1 , 12 $ 1 2
3
```

CONNECTIONS

A *directed graph* is a collection of *nodes* with connections or *arcs* specified between certain pairs of nodes. It can be used to specify things such as the precedences in a set of processes (stuffing of envelopes must precede sealing), or the structure of a *tree*.

The connections can be specified by a boolean *connection matrix* instead of by arcs, and the connection matrix can be determined from the list of arcs.

The connection matrix is very convenient for determining various properties of the graph, such as the *in-degrees* (number of arcs entering a node), the *out-degrees*, immediate descendants, and the *closure*, or connection to every node reachable through some path.

CONNECTIONS (Arcs)

```
f=.'372557155526123774727 4'

t=.'560262760733217042300 3'
   d=.'01234567'

   arcs=.(d i. f),:(d i. t)

   15{."1 arcs
3 7 2 5 5 7 1 5 5 5 2 6 1 2 3
5 6 0 2 6 2 7 6 0 7 3 3 2 1 7

   [n=.arcs{nodes=.'ABCDEFGH'
DHCFFHBFFFCGBCDHHEHCHE
FGACGCHGAHDDCBHAECDAAD
     2 11$<"1|:n
```

DF	HG	CA	FC	FG	HC	BH	F
GD	BC	CB	DH	HA	HE	EC	H

etc.

CONNECTIONS (Connection matrix)

```
   q=.i.@(,~@[)e.]+/ . *,&1@[

   cmFROMarcs=. [ q |:@]

   cm=. 8 cmFROMarcs arcs
   cm
0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 1
1 1 0 1 0 0 0 0
0 0 0 0 0 1 0 1
0 0 1 1 0 0 0 0
1 0 1 0 0 0 1 1
0 0 0 1 0 0 0 0
1 0 1 1 1 0 1 0

   ] indegrees=. +/cm
3 1 4 4 1 1 2 3
   +/+/cm
19
```

CONNECTIONS (Family)

```
   cm
0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 1
1 1 0 1 0 0 0 0
0 0 0 0 0 1 0 1
0 0 1 1 0 0 0 0
1 0 1 0 0 0 1 1
0 0 0 1 0 0 0 0
1 0 1 1 1 0 1 0
   points=. 1 0 0 0 0 0 0 1
   points +./ . *. cm
1 0 1 1 1 0 1 0
   points+.points+./ . *.cm
1 0 1 1 1 0 1 1

   immfam=. [ +. [ +./ . *. ]
   points immfam cm
1 0 1 1 1 0 1 1
   fam=. [&immfam ^: _
   points fam cm
1 1 1 1 1 1 1 1
```

CONNECTIONS (Adjacency)

```
   d=. #: i. 2^3
   d
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
   adj=.1: = |: +/ . ~: ]
   ]e=. adj d
0 1 1 0 1 0 0 0
1 0 0 1 0 1 0 0
1 0 0 1 0 0 1 0
0 1 1 0 0 0 0 1
1 0 0 0 0 1 1 0
0 1 0 0 1 0 0 1
0 0 1 0 1 0 0 1
0 0 0 1 0 1 1 0
   e{' *'
 ** *
*  * *
*  *  *
 **    *
*    **
 *  *  *
  * *  *
   * **
```

CONNECTIONS (Closure)

```
   ]cm2=. |. = i. 8
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
   points fam cm2
1 1 1 1 1 1 1 1
   cm2 fam cm2
1 0 0 0 0 0 0 1
0 1 0 0 0 0 1 0
0 0 1 0 0 1 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 1 0 0 1 0 0
0 1 0 0 0 0 1 0
1 0 0 0 0 0 0 1
```

SETS (Propositions)

```
   [ a=. 2%~ i. 11
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5
   (2&<: *. <&5) a
0 0 0 0 1 1 1 1 1 1 0
   ((2&<: *. <&5) a) # a
2 2.5 3 3.5 4 4.5
   ((2&<: *. <&5) # ]) a
2 2.5 3 3.5 4 4.5
   (] #~ 2&<: *. <&5) a
2 2.5 3 3.5 4 4.5
   int=. = <.
   int a
1 0 1 0 1 0 1 0 1 0 1
   ((2&<: *. int) a) # a
2 3 4 5
   (] #~ 2&<: *. int) a
2 3 4 5
   (#~ 2&<: *. int) a
2 3 4 5
```

```
   i=.i.8  [  p=. 2 3 5 7 11
   belongsto=. +./@(=/)~

   i belongsto p
0 0 1 1 0 1 0 1
   e=. belongsto
   p e i
1 1 1 1 0

   c=. -.@v=. e&'aeiou'
   alph=.  'abcdefghijklmno'
   alph=. alph,'pqrstuvwxyz'

   (v alph)#alph
aeiou

   (#~ c) alph
bcdfghjklmnpqrstvwxyz

   alph-.'aeiou'
bcdfghjklmnpqrstvwxyz
```



```
   (even=. 0&=@(2&|))a=.i. 16
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
p=.2&=@(+/)@(0:=]|~>:@(i.@]))
   prime=. p"0
   prime a
0 0 1 1 0 1 0 1 0 0 0 1 0 1 0 0
   a #~ prime a
2 3 5 7 11 13
   a#~(prime*.even)a
2
   a#~(prime>even)a
3 5 7 11 13
   triple=.0&=&(3&|)
   q=. even+.triple
   (q a) # a
0 2 3 4 6 8 9 10 12 14 15
r=. prime +. even *. triple
   (r a) # a
0 2 3 5 6 7 11 12 13
```

CONTROLLED ITERATION

Programming languages commonly use *control structures* for controlled iteration of a process (DO WHILE), and for the application of one of several processes (CASE STATEMENT). In J, the *power* conjunction provides iteration for a fixed number of times specified by a noun (as in **f^:4**), and for a variable number of times determined by a verb **g** (in **f^:g**).

Cases are controlled by the verb right argument of the *agenda* conjunction, as in **n@.v** to select one of the functions used to form the *gerund* **n** in an expression such as **n=. 1&o.`^.`^**

Self-reference to the verb being defined is provided by **$:** (as in **n=. 1:`(*$:@<:)**), and **n@.v** can therefore provide recursive definition, without naming the resulting verb.

Since much iteration occurs automatically (as in **list+list** and **list+/list** and **+/list**), we will illustrate controlled iteration by Newton's method for a single root of a polynomial, and by its **n**-dimensional analog (Kerner's method) for all roots of a polynomial of degree **n**.

```
   poly=. #.~&|."1 0
   1 3 3 1 poly 3 4 5
64 125 216
   c=. 12 _10 2
   deriv=. }.@(* i.@#)
   deriv c
_10 4
   n=. ]-poly%deriv@[poly]
   c (newton=. n) approx=.2.4
1.2
   c newton c newton approx
1.75385
   c&newton ^: 0 1 2 3 approx
2.4 1.2 1.75385 1.9594
   root=. c&newton ^:_ approx
   root NB. _ is infinity.
2
   c poly root
0
```

CONTROLLED ITERATION

(Kerner's Method for all roots)

```
   norm=. % {:
   norm c=. 12 _10 2
 6 _5 1
   (init=. r.@}.@i.@#@]) c
0.540302j0.841471 _0.416147j0.909297
deriv=.(*/@(-"0 1) 1&(]\.))@]
   kerner=.]-poly % deriv
 r=.(norm c)&kerner^:_ init c
   ]roots=. r
2j_1.97994e_27 3j1.98001e_27
   | c poly roots
8.67362e_19 3.96002e_27
   d=.%!i.6
r=.(norm d)&kerner^:_ init d
   +. (/:|) roots=. r
_2.18061 _3.76158e_37
_1.6495      _1.69393
_1.6495       1.69393
0.239806      3.12834
0.239806     _3.12834
```

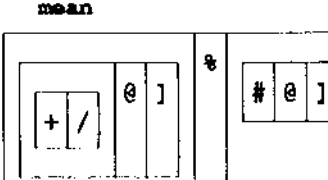
EXPLICIT PROGRAMS

Programming languages often define functions (procedures) by one or more sentences in which the arguments are referred to explicitly. We now illustrate such an explicit scheme where sentences are represented as character arrays, and left and right arguments are referred to by `x.` and `y.` Thus:

```
   s=. '(+/y.) % # y.'
   (MEAN=. s : '') 3 4 5
4
```

A one-sentence definition may be converted to equivalent tacit form:

```
   mean=. s : 20
   mean
```



etc.

RECURSIVE DEFINITION

(Using Tacit Definition)

```
   g=. 5&* ` ! ` -
   agenda=. 3&|
   case=. g @. agenda " 0
   case b=. 0 1 2 3 4 5 6 7
0 1 _2 15 24 _5 30 5040
   $g
3
   (|.g)@.agenda"0 b
0 1 10 _3 24 25 _6 5040

 factorial=. 1:`(*$:@<:)@.*"0
   factorial i.9
1 1 2 6 24 120 720 5040 40320
   1:`(*$:@<:)@.*"0 i. 9
1 1 2 6 24 120 720 5040 40320
```

EXPLICIT PROGRAMS (Simple)

```
   root=.'y. ^ %2':'y. ^ %x.'
   (root 64) , (3 root 64)
8 4
   rPr=. '% y.':'x. + % y.'
   3 rPr 4
3.25
   rPr / 1 2 2 2 2 2 2
1.4142
   rPr/ \ 1 2 2 2 2
1 1.5 1.4 1.41667 1.41379
   rPr/ \ 3 7 15
3 3.14286 3.14151
   triple=. '3*y.':''
   triple i.5
0 3 6 9 12
   3 triple 6
domain error
   tr=. '3*y.' : *
   tr i. 5
0 3 6 9 12
   3 5 7 tr i. 3
0 5 14
```

EXPLICIT PROGRAMS (Conditional)

```
   p=. '$.=. 1+y.<0'
   q=. 'y. ^ %2'
   r=. '''DOMAIN ERROR'''
   conditional=. (p;q;r) : ''
   conditional -49
DOMAIN ERROR
   conditional 49
7
tozero=.(p;'y.-1';'y.+1') :''
   tozero 3
2
   tozero _3
_2
   tozero "0 (_2 _1 0 1 2 3)
_1 0 _1 0 1 2
   tozero
┌───────────┬─┬┐
│$.=. 1+y.<0│:││
│y.-1       │ ││
│y.+1       │ ││
└───────────┴─┴┘
```

EXPLICIT PROGRAM (Recursive)

```
   a=. '$.=. 2-0=y.' ; '1'
   b=. 'y. * factorial y.-1'
   factorial=. (a,<b) : ''
   factorial 5
120
d=.'(r,0)+0,r=.binomial y.-1'
   binomial=. (a,<d) : ''
   binomial 4
1 4 6 4 1
   f=.'r,+/_2{.r=.fib y.-1'
   fib=. (a,<f) : ''
   fib 10
1 1 2 3 5 8 13 21 34 55 89
   +^:(i.11)~1
1 1 2 3 5 8 13 21 34 55 89
   g=. '$.=. 2-0=x.' ; '1'
   h=. 'y.*x.%~x. outof&<:y.'
   outof=. '':(g,<h)
   outof"0/~i. 4
1 1 1 1
0 1 2 3
0 0 1 3
0 0 0 1
```


EXPLICIT PROGRAM (Iterative)

```
   a=. 'r=. 1 [ $.=. y. # 1'
   b=. 'r=. r * 1+ # $.'
   factorial=. (a;b) : ''
   factorial 5
120
   factorial"0 i. 6
1 1 2 6 24 120
   > a;b
r=. 1 [ $.=. y. # 1
r=. r * 1+ # $.
   c=. 'r=. (0,r) + (r,0)'
   binomials=. (a;c) : ''
   binomials 4
1 4 6 4 1
fib=.(a;'r=.r,+/_2{.r') : ''
   fib 10
1 1 2 3 5 8 13 21 34 55 89
   d=. 'r=. 1 [ $.=. x. # 1'
e=. 'r=.(r*1+y.=.y.-1)%1+#$.'
   outof=. '':(d;e)
   3 outof 5
10
```

EXPLICIT PROGRAM (Recursive)

```
   [ a=.3 3$'abcdefghi'
abc
def
ghi
   (f=. -."1 0~@(i.&#)) a
1 2
0 2
0 1
<"2 (m=. minors=.f { 1&}."1)a
┌──┬──┬──┐
│ef│bc│bc│
│hi│hi│ef│
└──┴──┴──┘
   p=.'$.=. 1+1=#y.'
q=.'(0{"1 y.)-/ .*det"2 m y.'
   ]b=. 3 3$1 6 4,4 1 0,6 6 8
1 6 4
4 1 0
6 6 8
(det=.(p;q;'0{,y.') : '') b
_112
   (-/ . * b) , (+/ . * b)
_112 320
```

EXPLICIT PROGRAMS

(Recursive)

```
   a=.'$.=.1+0<n=.x.-1'
   b=.',:2{.y.'

   H=. '' : (a;b;c,'n H|.y.')
   hanoi=. H
   2 hanoi 'ABC'
AC
AB
CB
   |: 4 hanoi 0 1 2
0 0 2 0 1 1 0 0 2 2 1 2 0 0 2
2 1 1 2 0 2 2 1 1 0 0 1 2 1 1
   |: 'ABC'{~ 4 hanoi 0 1 2
AACABBAACCBCAAC
CBBCACCBBAABCBB
   c=.'r=.0#$.=.y.#1+n=.0'
   d=.'r=.r,(n=.1+n),r'
   h=.(c;d) : ''
   h 4
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
   h 3
1 2 1 3 1 2 1
```

DEFINED CONJUNCTIONS

```
   ]a=. >:i.8
1 2 3 4 5 6 7 8
   modulo=. 'y.&|@(x.f.)' : 2
   8 + modulo 4 (7)
3
   + modulo 4 /~ a
2 3 0 1 2 3 0 1
3 0 1 2 3 0 1 2
0 1 2 3 0 1 2 3
1 2 3 0 1 2 3 0
2 3 0 1 2 3 0 1
3 0 1 2 3 0 1 2
0 1 2 3 0 1 2 3
1 2 3 0 1 2 3 0
   * modulo 4 /~ a
1 2 3 0 1 2 3 0
2 0 2 0 2 0 2 0
3 2 1 0 3 2 1 0
0 0 0 0 0 0 0 0
1 2 3 0 1 2 3 0
2 0 2 0 2 0 2 0
3 2 1 0 3 2 1 0
0 0 0 0 0 0 0 0
```

DEFINED ADVERBS

```
   ]b=.>: i.4
1 2 3 4
   scan=. /\
   + scan b
1 3 6 10
   * scan b
1 2 6 24
   inv=. ^:_1
   ^ inv b
0 0.693147 1.09861 1.38629
   ^. b
0 0.693147 1.09861 1.38629
   3&* inv b
0.333333 0.666667 1 1.33333
   slope=.'[%~+-&(x.f.)]' : 1
   0.1 ^ slope 1 2 3
2.85884 7.77114 21.1241
   ^ 1 2 3
2.71828 7.38906 20.0855
   1e_6 ^ slope 1 2 3
2.71828 7.38906 20.0855
```

DEFINED CONJUNCTIONS

```
tacitmod=.'y.&|@(x.f.)' : 22
   ^ tacitmod 4 /~a
1 1 1 1 1 1 1 1
2 0 0 0 0 0 0 0
3 1 3 1 3 1 3 1
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
2 0 0 0 0 0 0 0
3 1 3 1 3 1 3 1
0 0 0 0 0 0 0 0
   tacitmod
+-----------------+-+-----------+
|+--+-------+     |@|+--+--+    |
||].|+-+-+  |     | ||[.|f.|    |
||  ||&|||  |     | |+--+--+    |
||  |+-+-+  |     | |           |
|+--+-------+     | |           |
+-----------------+-+-----------+
  modulo
+-----------+-+-+
|y.&|@(x.f.)|:|2|
+-----------+-+-+
```

```
   x=.1 2 3 4 5 6 7
   x^2
1 4 9 16 25 36 49
   x^3
1 8 27 64 125 216 343
   (4*x^2) + (_3*x^3)
1 _8 _45 _128 _275 _504 _833

   2 3 ^~/ x
1 4  9 16  25  36  49
1 8 27 64 125 216 343
   4 _3 +/ . *2 3 ^~/x
1 _8 _45 _128 _275 _504 _833

   e=. 0 1 2 3 4
   vandermonde=. e ^~/ x
   vandermonde
1  1  1   1   1    1    1
1  2  3   4   5    6    7
1  4  9  16  25   36   49
1  8 27  64 125  216  343
1 16 81 256 625 1296 2401
```



```
   c=. 4 2 _3 2 1
   vandermonde
1  1  1   1   1    1    1
1  2  3   4   5    6    7
1  4  9  16  25   36   49
1  8 27  64 125  216  343
1 16 81 256 625 1296 2401
   c +/ . * vandermonde
6 28 118 348 814 1636 2958
poly=.[+/ .*|:@(]^/ i.@(#@[))
   c poly x
6 28 118 348 814 1636 2958
```

Stirling Numbers

```
   S1=. ^!._1/~@i.%.^/~@i.
   ".@(0&":)&.>@(S1 ; %.@S1)5
┌────────────┬─────────┐
│1 0  0  0  0│1 0 0 0 0│
│0 1 _1  2 _6│0 1 1 1 1│
│0 0  1 _3 11│0 0 1 3 7│
│0 0  0  1 _6│0 0 0 1 6│
│0 0  0  0  1│0 0 0 0 1│
└────────────┴─────────┘
```

INVERSES AND DUALITY

```
   cFf=. -&32 * (5%9)"0
   fFc=. 32&+@(*&1.8)
   dc=. 40 -~ 20 * i. 8
   dc
_40 _20 0 20 40 60 80 100

   fFc dc
_40 _4 32 68 104 140 176 212

   cFf fFc dc
_40 _20 0 20 40 60 80 100
   % % 1 2 3
1 2 3
   log=.10&^.
   invlog=.10&^
   log y=. 24 4 75
1.38021 0.60206 1.87506
   +/ log y
3.85733
   invlog +/ log y
7200
   */y
7200
```

INVERSES AND DUALITY

```
   r=. 2 3 4  [  s=. 2 4 5

   invlog (log r) + (log s)
4 12 20
   r * s
4 12 20

   ^ (^. r) + (^. s)
4 12 20
   r +&.^. s
4 12 20

   r +&.% s
1 1.71429 2.22222
   % (%r) + (%s)
1 1.71429 2.22222

   +&.% / r
0.923077
   % +/ % r
0.923077
```

```
   f=. +&3
   g=. -&3
   [ x=. i. 4
0 1 2 3
   f x
3 4 5 6
   !f x
6 24 120 720
   g!f x
3 21 117 717
   !&.f x
3 21 117 717
   !&.(+&3) x
3 21 117 717
   !&.(*&2) x
0.5 1 12 360
```

PERMUTATIONS

```
   a=. 'ABCDEF'
   p=. 2 3 5 1 4 0
   p{a
CDFBEA
   p C. a
CDFBEA
   ]c=. C. p
┌───┬─┬─────┐
│3 1│4│5 0 2│
└───┴─┴─────┘
   c C. a
CDFBEA
   A. p
309
   A. c
309
   309 A. a
CDFBEA
   0 A. 0 1 2
0 1 2
   1 A. 0 1 2
0 2 1
   j=.i.5
   (j A. i. 3);(j A. 'abc')
┌─────┬───┐
│0 1 2│abc│
│0 2 1│acb│
│1 0 2│bac│
│1 2 0│bca│
│2 0 1│cab│
└─────┴───┘
```

UTILITIES

```
   names=. 4!:1
   save=. 2!:2
   copy=. 2!:4
   names 2        List of pronouns
   names 3        List of proverbs
   save <'abc'    Save named objects
   erase=. 4!:55     in file abc
   off=. 0!:55
erase names 2     Erase pronouns
off ''            End session
Control D         End session
5!:2 <'abc'       Representation of
          verb abc (in Display form )
5!:4 <'abc'  Tree form
```

Verbs for bordering verb tables:

```
   over=.({.;}.)@":@,
   by=.' '&;@,.@[,.]
```

CUT

```
 ]rt=.1|.t=.'/Onward/he said'
Onward/he said/
   < ;. 1 text=. t
┌───────┬────────┐
│/Onward│/he said│
└───────┴────────┘
   < ;._1 text
┌──────┬───────┐
│Onward│he said│
└──────┴───────┘
   # ;. _1 text
6 7
   < ;._2 rt
┌──────┬───────┐
│Onward│he said│
└──────┴───────┘
   i. 4 5
0  1  2  3  4
5  6  7  etc.
   ]q=.0 1 0 1 <;.1 i. 4 5
┌──────────────┬───────────
│ 5  6  7  8  9│15 16 17 1
│10 11 12 13 14│       etc.
└──────────────┴───────────
   1{q
┌──────────────┐
│15 16 17 18 19│
└──────────────┘
```

EXERCISES

2.1 Enter the following sentences on the computer, observe the results, give suitable names to any new primitives (such as * and +. and *.), and comment on their behaviour.

```
a=.0 1 2 3
b=.3 2 1 0
a+b
a*b
a-b
a%b
a^b
a^.b
a<b
a>b
(a<b)+(a>b)
(a<b)+.(a>b)
```

Compare your comments with the following:

a) Negative `3` is denoted by `_3`. The underbar `_` is part of the representation of a negative number in the same sense that the decimal point is part of the representation of one-half when written in the form `0.5`, and the negative sign `_` must not be confused with the symbol used to denote subtraction (i.e., `-`).

b) Division (`%`) by zero yields infinity, denoted by the underbar alone.

c) Log of `2` to the base `1` is infinite, and log of `0` to the base `3` is negative infinity (`__`).

d) Since the relation `5<7` is true, and the result of `5<7` is `1`, it may be said that true and false are represented by the ordinary integers `1` and `0`. George Boole used this same convention, together with the symbol `+` to represent the *boolean* function *or*. We use the distinct representation `+.` to avoid conflict with the analogous (but different) *addition* (denoted by `+`).

2.2 Following the example `Min=. <.` , invent, assign, and use names for each of the primitives encountered thus far.

3.1 Enter the following sentences (and perhaps related sentences using different arguments), observe the results, and state what the two cases (*monadic* and *dyadic*) of each function do:

```
a=. 3 1 4 1 5 9
b=. 'Canada'
#a
1 0 1 0 1 3 # a
1 0 1 0 1 3 # b
/: a
/:b
a /: a
a /: b
b /: a
b /: b
c=. 'can''t'
c
#c
c /: c
```

3.2 Make a summary table of the functions used thus far. Then compare it with the following table (in which a slash separates the monadic case from the dyadic, as in negation / addition:

+	· Add	· Or	
-	Negate · Subtract		
*	·Times	· And	
%	Reciprocal · Divide		
^	Exponential · Power	· Log	
<	· Less Than	· Lesser Of	
>	· Greater Than	· Greater Of	
=	·Equals	Is (Copula)	
i		Integers · Index Of	
$	Shape · Reshape		
/			Grade · Sort
#	Number of items · Replicate		

3.3 Try to fill some of the gaps in the table of Exercise 3.2 by experimenting on the computer with appropriate expressions. For example, enter `^. 10` and `^. 2.71828` to determine the missing (monadic) case of `^.` and enter `%: 4` and `%: -4` and `+%: -4` to determine the case of `%` followed by a colon.

However, do not waste time on matters (such as, perhaps, complex numbers or the *boxed* results produced by the monad `<`) that are still beyond your grasp; it may be better to return to them after working through later lessons. Note that the effects of certain functions become evident only when applied to arguments other than integers. For example, try `<.1 2 3 4` and `<.3.4 5.2 3.6` to determine the effect of the monad `<.` .

3.4 If `b=.3.4 5.2 3.6` , then `<.b` yields the argument `b` rounded *down* to the nearest integer. Write and test a sentence that rounds the argument `b` to the *nearest* integer

ANSWER: `<.(b+0.5)` or `<.b+0.5` or `<.b+1r2`

3.5 Enter `2 4 3 $ i. 5` to see an example of a *rank* `3` *array* or *report* (for two years of four quarters of three months each).

3.6 Enter `?9` repeatedly and state what the function `?` does. Then enter `t=. ?3 5 $ 9` to make a table for use in further experiments.

ANSWER: `?` is a (pseudo-) random number generator; `?n` produces an element from the population `i.n`

4.1 Enter `d=. i.5` and the sentences `st=. d-/d` and `pt=. d^/d` to produce function tables for subtraction and power.

4.2 Make tables for further functions from previous lessons, including the relations `<` and `=` and `>` and the *lesser-of* and *greater-of*.

4.3 Apply the verbs `|.` and `|:` to various tables, and try to state what they do.

4.4 The *transpose* function `|:` changes the subtraction table, but appears to have no effect on the multiplication table. State the property of those functions whose tables remain unchanged when transposed.

ANSWER: They are commutative

4.5 Enter `d by d over d!/d` and state the definition of the dyad `!` .

ANSWER: `!` is the *binomial coefficient* or *outof* function: `3!5` is the number of ways that three things can be chosen from five.

5.1 In math, the expression $3x^4+4x^3+5x^2$ is called a *polynomial.* Enter:

```
x=. 2
(3*x^4)+(4*x^3)+(5*x^2)
```

to evaluate the polynomial for the case where `x` is `2`.

5.2 Note that the hierarchy among functions used in math is such that no parentheses are necessary in writing a polynomial. Write an equivalent sentence using no parentheses.

ANSWER: `+/3 4 5 * x ^ 4 3 2` or (first assigning names to the *coefficients* `3 4 5` and the *exponents* `4 3 2`), as `+/c*x^e`

6.1 Enter `5#3` and similar expressions to determine the definition of the dyad `#` and then state the meaning of the following sentence:

```
(# # >./) b=. 2 7 1 8 2
```

ANSWER: `#b` repetitions of the maximum over `b`

6.2 Cover the comments on the right, write your own interpretation of each sentence, and then compare your statements with those on the right:

`(+/ % #) b`	Mean of `b`
`(# # +/ % #) b`	`(n=.#b)` repetitions of mean
`+/(##+/%#) b`	Sum of `n` means
`(+/b)=+/(##+/%#) b`	Tautology
`(*/b)= */(###%:*/) b`	The product over `b` is the product over `n` repetitions of the geometric mean of `b`.

7.1 Enter `AT=. i. +/ i.` and use expressions such as `AT 5` to determine the behaviour of the program `AT`.

7.2 Define and use similar function tables for other dyadic functions.

7.3 Define the programs:

```
tab=. +/
ft=. i. tab i.
test1=. ft = AT
```

Then apply `test1` to various integer arguments to test the proposition that `ft` is equivalent to `AT` of Exercise 7.1, and enter `ft` and `AT` alone to display their definitions.

7.4 Define the program `aft=. ft f.` and use `test2=. aft = ft` to test their equivalence. Then display their definitions and state the effect of the adverb `f.` .

ANSWER: The adverb `f.` *fixes* the verb to which it applies, replacing each name used by its definition.

7.5 Redefine `tab` of Exercise 7.3 by entering `tab=. */` and observe the effects on the function `ft` and its *fixed* alternative `aft`.

7.6 Define `mean=. +/ % #` and state its behaviour when applied to a *table*, as in `mean t=. (i. !/ i.) 5`.

ANSWER: The result is the average over the *rows* of a table argument.

7.7 Write an expression for the mean over the *columns* of `t`.

ANSWER: `mean |: t`

8.1 The verb `#` is used dyadically in the definition of the program `IN29`. Enter expressions such as `(j=. 3 0 4 0 1) # i.5` to determine the behaviour of `#`, and state the result of `#j#i.5`.

ANSWER: `+/j`

8.2 Cover the answers on the right and apply the following programs to lists to determine (and state in English) the purpose of each:

`test1=. >&10 *. <&100`	Test if in 10 to 100
`int=. ] = <.`	Test if integer
`test2=. int *. test1`	Test if integer and in 10 to 100
`test3=. int +. test1`	Test if integer or in 10 to 100
`sel=. test2 # ]`	Select integers in 10 to 100

8.3 Cover the program definitions on the left of the preceding exercise, and make new programs for the effects stated on the right.

8.4 Review the use of the *fix* adverb in Exercises 7.4-5, and experiment with its use on the programs of Exercise 8.2.

9.1 Cover the comments on the right, and state the effects of the programs. Then cover the programs and rewrite them from the English statements:

`mc=. (+/%#)@|:`	Mean over columns of table
`f=. +/@*:`	Sum of squares of list
`g=. %:@f`	Geometric length of list
`h=.{&' *'@(</)`	Map of comparison (dyad)
`k=. i. h i.`	Map (monad)
`map=. {&'+-* %#$'`	7-character map
`MAP=. map@(6&<.)@<.`	Extended domain of `map`
`add=. MAP@ (i.+/i.)`	Addition table map

10.1 Experiment with a revised version of the program `MAP` of Exercise 9.1, using the *remainder* or *residue* dyad (`|`) instead of the *minimum* (`<.`), as in `M=. map@(6&|)@<.` and compare its results with those of `MAP`.

10.2 Experiment with the programs `sin` and `sind` defined in this lesson.

10.3 Write programs using various new primitives found in the vocabulary of Appendix E.

10.4 Update the table of notation prepared in Exercise 3.2.

11.1 Enter and experiment with the programs defined in this lesson.

12.1 The square function `*:` is the inverse of the square root function `%:` and `%:^:_1` is therefore equivalent to `*:` . Try to find other inverse pairs among the primitive functions in the summary table of the dictionary.

13.1 These exercises are grouped by topic and organized like the lesson, with programs that are first to be read and then to be rewritten. However, a reader already familiar with a given topic might begin by writing.

A. Properties of numbers

`pn=. >:@i.` Positive numbers (e.g. `pn 9`)

`rt=. pn |/ pn` Remainder table

`dt=. 0&=@rt` Divisibility table

`nd=. +/@dt` Number of divisors

`prt=. 2&=@nd` Prime test

`prsel=. prt # pn` Prime select

`N=. >:@pn` Numbers greater than 1

`rtt=. ,@(N */ N)` Ravelled times table

`aprt=. -.@(N e. rtt)` Alternate test and selection (primes do not occur in the * table for `N`)

`apsel=. aprt # N`

B. Coordinate Geometry

Do experiments on the vector (or *point)* `p=. 3 4` and the triangle represented by the table `tri=. 3 2$ 3 4 6 5 7 2`

`L=. %:@(+/)@*:` Length of vector

`LR=.L"1` Length of rows in table (See *rank* in the dictionary or in Lesson 21)

`disp=. ] - 1&|.` Displacement between rows in a table

`LS=. LR@disp` Lengths of sides of figure

`sp=. -:@(+/)@LS` Semiperimeter (try `sp tri`)

`H=. %:@(*/)@(sp,sp-LS)` Heron's formula for triangle area

`det=. -/ . *` Determinant (See dictionary)

`SA=. det@(,.&0.5)` Signed area; positive if vertices are in counterclockwise order when plotted

`sa=.det@(],.%@!@<:@#)` General signed volume; try it on the tetrahedron `tet=. ?4 3$9` as well as on the triangle `tri`

14.1 Using the programs defined in Lesson 13, experiment with the following expressions:

```
5.2 ": d=. %: i.12
5.2 ":,.d
fc=. 5.2&":@,.
fc d
20 (fc@h3 ,. h5) d
20 (fc@h3 ,. '|'&,.@h5) d
plot=. fc@h3,.'|'&,.@h5
20 plot d
```

15.1 Define programs analogous to `sum=.+/\` for progressive products, progressive maxima, and progressive minima.

15.2 Treat the following programs and comments like those of Lesson 13, that is, as exercises in reading and writing. Experiment with expressions such as `c pol x` and `c pp d` and `(c pp d) pol x` with `c=. 1 3 3 1` and `d=. 1 2 1` and `x=.i. 5`. See the dictionary or Lesson 21 for the use of rank (`"`):

```
pol=. +/@([*]^i.@#@[)"1 0   Polynomial
pp=. +//.@(*/)              Polynomial product
pp11=. 1 1&pp               Polynomial product by 1 1
pp11 d
pp11^:5 (1)
ps=. +/@,:                  Polynomial sum
```

16.1 Experiment with, and explain the behaviour of, the adverbs `pow=. ^&` and `log=. &^`.

16.2 State the significance of the following expressions, and test your conclusions by entering them:

```
+/~ i=. i. 6     Addition table
ft=. /~          Function table adverb
+ ft i           Addition table
! ft i           Binomial coefficients
inv=. ^:_1       Inverse adverb
sub3=. 3&+ inv   Subtract-three function
sub3 i
```

17.1 Choose sentences such as `pp=. +//.@(*/)` from earlier exercises, enclose them in quotes, and observe the effects of word-formation (`;:`) upon them.

18.1 Experiment with the use of locatives.

19.1 Comment on the results of the following experiments:

```
roots=. '3%:y.' : 'x.%:y.'
ROOTS=. 3&%: : %:
fib=. '' : ('r=. y.';'$.=. x.#2';'r=. r,+/_2{.r')
6 fib 0 1
```

ANSWER: `roots` and `fib` are from Lesson 19; `ROOTS` shows the use of the conjunction `:` with verb arguments to specify the monadic and dyadic parts of the resulting function.

19.2 Define the following adverbs, and experiment with them in expressions such as `! h b=. i.7`

```
h=. '-:@(x.f.)' : 1
d=. '+:@(x.f.)' : 1
dh=. '+:@(x.f.)@-:' : 1
```

19.3 Using the program `pol` from Exercise 15.2, perform the following experiments and comment on their results:

```
g=. 11 7 5 3 2 & pol
e=. 11 0 5 0 2 & pol
o=.  0 7 0 3 0 & pol
(g = e + o) b=. i.6
(e = e@-) b
(o = -@o@-) b
```

ANSWER: The function `g` is the sum of the functions `e` and `o`. Moreover, `e` is an *even* function (whose graph is reflected in the vertical axis), and `o` is an *odd* function (reflected in the origin).

19.4 Enter the following explicit definition of the adverb `even` and perform the suggested experiments with it, using the functions defined in the preceding exercise:

```
even=. '-:@(x.f. + x.f.@-)' : 1
```

```
ge=. g even
(e = ge) b
(e = e even) b
```

19.5 Define an adverb `odd` and use it in the following experiments:

```
exp=. ^
sinh=. 5&o.
cosh=. 6&o.
(sinh = exp odd) b
(sinh =. exp .: -) b        The primitive odd adverb .: -
(cosh = exp even) b
(exp = exp even + exp odd) b
```

19.6 These experiments involve complex numbers, and should perhaps be ignored by anyone unfamiliar with them:

```
sin=. 1&o.
cos=. 2&o.
(cos = ^@j. even) b
(j.@sin = ^@j. odd) b
```

20.1 Use the display of the tacit definition of `MEAN` in Lesson 20 to enter a tacit definition of an equivalent function called `M`.

ANSWER: `M=. +/@] % #@]`

20.2 Simplify the definition of `M` of the preceding exercise to produce an equivalent tacit definition called `m`.

ANSWER: `m=. +/ % #`

20.3 Use the display of the tacit definition of the conjunction `u` in Lesson 20 as a guide in entering the tacit definition of an equivalent conjunction to be called `U`, and compare it with the simplified form used in defining `UNDER` in Lesson 20.

ANSWER: `U=. ((].f.)(^:_1))@(([.f.)&(].f.))`

20.4 Enter the definition of the adverb `h` of Exercise 19.2 in two steps as follows:

```
s=. '-:@(x.f.)'
h=. s : 1
```

Then enter `hc=. s : 22` to obtain the tacit definition of a related *conjunction*, and confirm that `! hc ] a=. i.6` is equivalent to `! h a`. Then define the corresponding *adverb* `H=. hc ]` and use it in the expression `! H a` and, finally, display all the entities defined.

21.1 Observe the results of the following uses of the monads produced by the rank conjunction, and comment on them:

```
a=. i. 3 4 5
<"0 a
<"1 a
<"2 a
<"3 a
< a
<"_1 a
<"_2 a
mean=. +/ % #
mean a
mean"1 a
mean"2 a
```

ANSWER: `<"k` applies `<` to each cell of rank `k`, with `<"(#$a) a` being equivalent to `<a`. Moreover, a negative value of `k` specifies a *complementary rank* that is effectively `|k` less than the rank of the argument `a`.

21.2 Use the results of the following experiments to state the relation between the conjunctions `@` (*Atop*) and `@:` (*At*), and compare your conclusions with the dictionary definition:

```
(g=. <"2) a=. i. 3 4 5
|. @: g a
|. @ g a
|: @: (<"1) a
|: @ (<"1) a
```

ANSWER: The rank of the function `|. @: g` is itself infinite and `|.` therefore applies to the entire list result of `g a`,

consequently reversing it. On the other hand, the function `f @ g` *inherits* the rank of `g`, and `|.` therefore applies individually to the atoms produced by `g`, producing no effect.

21.3 Use the results of the following experiments to comment on the use of the rank conjunction in dyads:

```
b=. 'ABC'
c=. 3 5 $ 'abcdefghijklmno'
c
b,c
b ,"0 1 c
b ,"1 1 c
b ,"1 c
```

22.1 Define a function `f` such that `(x=. 4) f c=. 1 3 3 1` yields the result used as the argument to ``+`*/`` in Horner's method in Lesson 22.

ANSWER: `f=. }.@,@,.`

23.1 Use the following as exercises in reading and writing:

``f=.1:`(+//.@(,:~)@($:@<:))@.*``	Binomial coefficients
`<@f"0 i.6`	Boxed binomials
``g=.1:`(([,+/@(_2&{.))@$:@<:)@.*``	Fibonacci sequence

24.1 Use the function `BR` of Lesson 24 to find the roots of various functions, such as `f=. 6&-@!`

24.2 Experiment with the function `fn=. +/\` (which produces the *figurate numbers* when applied repeatedly to `i.n`), and explain the behaviour of the function `fn^:(?@(3&[))`

25.1 Use the display of `s : 12` in Lesson 25 as a guide in defining an equivalent conjunction `C`, and compare the resulting definition with the simpler definition used for `cj` in Lesson 25.

26.1 Use the following as exercises in reading and writing (try the programs on `a=.'abcdef'` and `b=.i. 6` and `c=. i. 6 6`):

`f=. 1&A.`	Interchange last two items
`g=. 3&A.`	Rotate last three items
`h=. 5&A.`	Reverse last three items
`i=. <:@!@[ A. ]`	`k i a` reverses last `k` items.

26.2 Experiment with the following expressions and others like them to determine the rules for using "abbreviated" arguments to `C.` and compare your conclusions with the dictionary definitions:

```
2 1 4 C. b=.i.6
(<2 1 4) C. b
(3 1;5 0) C. b
```

26.3 Make a program `ac` that produces a table of the cycle representations of all permutations of the order of its argument, as in `ac 3`.

ANSWER: `ac=. C.@(i.@! A. i.)`

27.1 For each of the following functions, determine the matrix `M` such that `M (mp=. +/ . *) N` is equivalent to the result of the function applied to the matrix `N`, and test it for the case `N=. i. 6 6`:

```
|.
-
+:
(4&*-2&*@|.)
2&A.
```

28.1 Use the following as exercises in reading and writing. Try using arguments such as `a=. 2 3 5 7` and `b=. 1 2 3 4` and `c=. <@i."0 i. 3 4`:

`f=. +&.^.` Multiplication by addition of natural logs

`g=. +&.(10&^.)`	Multiplication using base-10 logs
`h=. *&.^`	Addition from multiplication
`i=. |.&.>`	Reverse each box

`j=. +/&.>`	Sum each box
`k=. +/&>`	Sum each box and leave open

29.1 Predict and test the results of the following expressions:

```
*/''
<./''
>./''
>./0 4 4 $ 0
+/ . */ 0 4 4 $ 0
+&.^./
```

30.1 Experiment with the verbs of Lesson 30, and consult their definitions in the dictionary.

30.2 Experiment with the dyad `{@;` and give the term used to describe it in mathematics.

ANSWER: Cartesian product

30.3 Test the assertion that the monads `%:` and `(%:@~. +/ . * =)` are equivalent, and state the utility of the latter when applied to a list such as `1 4 1 4 2` that has repeated elements.

ANSWER: The function `%:` (which could be a function costly to execute) is applied only to the distinct elements of the argument (as selected by the *nub* function `~.`)

30.4 Experiment with the adverbs and conjunctions given in Lesson 30, and consult their dictionary definitions.

30.5 Comment on the following experiments before reading the comments on the right:

`a=. 2 3 5 [ b=. 1 2 4`	
`a (f=. *:@+) b`	Square of sum
`a (g=. +&*: + +:@*) b`	Sum of squares plus double product
`a (f=g) b`	Expression of the identity of the functions
`a (f-:g) b`	`f` and `g` in a *tautology* (whose result is
`taut=. f-:g`	*always* true; that is, `1`).

30.6 A phrase such as `f-:g` may be a tautology for the dyadic case only, for the monadic case only, or for both. Use the following

tautologies as reading and writing exercises, including statements of applicability (Dyad only, etc.):

`t1=. >: -: > +. =`	(Dyad only) The primitive `>:` is identical to greater than *or* equal
`t2=. <. -: -@>.&-`	(Both) Lesser-of is neg on greater-of on neg; Floor is neg on ceiling on neg
`t3=. <. -: >.&.-`	Same as `t2` but uses *under*
`t4=. *:@>: -: *: + +: + 1:`	(Monad) Square of `a+1` is square of `a` plus twice `a` plus `1`
`t5=. *:@>: -: #.&1 2 1"0`	Same as `t4` using polynom
`t6=. ^&3@>:-:#.&1 3 3 1"0`	Like `t5` for cube
`bc=. i.@>: ! ]`	Binomial coefficients
`t7=.(>:@]^[)-:(]#.bc@[)"0`	Like `t6` with `k&t7` for `k`th power
`s=. 1&o. [. c=. 2&o.`	Sine and Cosine
`t8=. s@+ -: (s@[ * c@])+(c@[ * s@])`	(Dyad) Addition
`t9=. s@- -: (s@[ * c@])-(c@[ * s@])`	and Subtraction formulas for sine
`det=. -/ . *`	Determinant
`perm=. +/ . *`	Permanent
`sct=. 1 2&o."0@(,"0)`	Sine and cosine tables
`t10=. s@- -: det@sct`	Same as `t9` but using the determinant of the sin and cos table
`t11=. s@+ -: perm@sct`	Like `t8` using the permanent
`S=. 5&o. [. C=. 6&o.`	Hyperbolic sine and cosine
`SCT=. 5 6&o."0@(,"0)`	Sinh and Cosh table
`t12=. S@+ -: perm@SCT`	Addition theorem for sinh
`SINH=. ^ .: -`	Odd part of exponential

`COSH=. ^ .. -`	Even part of exponential
`t13=. SINH -: S` exp	(Monad) Sinh is odd part of
`t14=. COSH -: 6&o.`	Cosh is the even part of exp
`sine=. ^&.j. .: -` `t15=. sine -: s`	Sine is the odd part of exp under multiplication by `0j1`

30.7 Comment on the following expressions before reading the comments on the right:

`g=. + > >.`	Test if sum exceeds maximum
`5 g 2` `5 g _2 _1 0 1 2`	True for positive arguments but not true in general
`f=. *.&(0&<)`	Test if both arguments exceed `0`
`theorem=. f <: g`	The truth value of the result of `f` does not exceed that of `g`. This may also be stated as "If `f` (is true) then `g` (is true)" or as "`f` implies `g`"

```
5 theorem _2 _1 0 1 2
```

DICTIONARY of J

J is a dialect of APL, a formal imperative language. Because it is imperative, a sentence in J may also be called an *instruction*, and may be *executed* to produce a *result*. Because it is formal and unambiguous it can be executed mechanically by a computer, and is therefore called a *programming language*. Because it shares the analytic properties of mathematical notation, it is also called an *analytic* language.

APL originated in an attempt to provide consistent notation for the teaching and analysis of topics related to the application of computers, and developed through its use in a variety of topics, and through its implementation in computer systems. Discussions of its design and evolution may be found in References [7-9].

A dictionary should not be read as an introduction to a language, but should rather be consulted in conjunction with other material such as the introduction in this text, and References [1- 6]. On the other hand, a dictionary should be used not only to find the meanings of individual words, but should also be studied to gain an overall view of the language.

I: ALPHABET and WORDS

The alphabet is standard ASCII, comprising *digits, letters* (of the English alphabet), the *underline* (used in names and numbers), the (single) *quote*, and others (which include the space) to be referred to as *graphics*. Alternative spellings for the *national use* characters (which differ from country to country) appear in Appendix A.

Numbers are denoted by digits, the underline (for negative signs and for infinity and minus infinity — when used alone or in pairs), the period (used for decimal points and *necessarily* preceded by one or more digits), the letter `e` (as in `2.4e3` to signify `2400` in exponential form), and the letter `j` to separate the real and imaginary parts of a complex number, as in `3e4j_0.56`. Also see Appendix B.

A numeric *list* or *vector* is denoted by a list of numbers separated by spaces. A list of ASCII characters is denoted by the list enclosed in quotes, a pair of adjacent quotes signifying the quote itself: `'can''t'` is the five-character abbreviation of the six-character word `'cannot'`.

Names (used for pronouns and other surrogates, and assigned referents by the copula, as in `prices=. 4.5 12`) begin with a letter and may continue with letters, underlines, and digits. A name that includes an underline is a *locative*, as discussed in Appendix C. A name with an appended colon is a *given name*; it can be assigned once only, unless erased for re-use.

A *primitive* may be (denoted by) any single graphic (such as + for *plus*) or by any such graphic modified by a following *inflection* (a period or colon), as in `+.` and `+:` for *or* and *nor*. A primitive may also be an inflected name, as in `e.` and `o.` for *membership* and *pi times*. Finally, any inflected primitive may be further inflected.

Appendix E shows the entire spelling scheme, and page footers show the ordering used. Word formation (`;:`) may be applied to literal lists to explore the rhematic rules.

II. GRAMMAR

The following sentences illustrate the six parts of speech:

```
   fahrenheit=. 50
   (fahrenheit-32)*5%9
10
   prices=. 3 1 4 2
   orders=. 2 0 2 1
   orders * prices
6 0 8 2
   +/orders*prices
16
   +/\1 2 3 4 5
1 3 6 10 15
   bump=. +&1
   bump prices
4 2 5 3
```

PARTS of SPEECH

`50 fahrenheit`	Nouns/Pronouns
`+ - * % bump`	Verbs/Proverbs
`/ \`	Adverbs
`&`	Conjunction
`( )`	Punctuation
`=.`	Copula

Verbs act upon nouns to produce noun results; the nouns to which a particular verb applies are called its *arguments*. A verb may have two distinct (but usually related) meanings according to whether it is applied to one argument (to its right), or to two arguments (left and right). For example, `2%5` yields `0.4`, and `%5` yields `0.2`.

An adverb acts on a single noun or verb to its *left*. For example, `+/` is a *derived* verb (which might be called *plus over*) that sums an ar-

gument list to which it is applied, and `*/` yields the product of a list. A conjunction applies to two arguments, either nouns or verbs.

Punctuation is provided by parentheses that specify the sequence of execution as in elementary algebra.

The word `=.` behaves like the copulas "is" and "are" and is read as such, as in "area is 3 times 4" for `area=. 3*4`. The name `area` thus assigned is a *pronoun* and, as in English, it plays the role of a noun. Similar remarks apply to names assigned to verbs, adverbs, and conjunctions. Entry of a name alone displays its value.

A. NOUNS

Nouns are classified in three independent ways: numeric or literal; open or boxed; arrays of various ranks. In particular, arrays of ranks 0, 1, and 2 are called *atom*, *list*, and *table*, or, in mathematics, *scalar*, *vector*, and *matrix*. Numbers and literals are represented as stated in Part I.

Arrays. A single entity such as `2.3` or `_2.3j5` or `'A'` or `'+'` is called an atom. The verb denoted by comma chains its arguments to form a list whose *shape* (given by the verb `$`) is equal to the number of atoms combined. For example:

```
   date=. 1,7,7,6
   $ date
4
   word=. 's','a','w'
   |. word                         |. date
was                             6 7 7 1
```

The verb `|.` used above is called *reverse*. The phrase `s$b` produces an array of shape `s` from the list `b`. For example:

```
   (3,4) $ date,1,8,6,7,1,9,1,7
1 7 7 6
1 8 6 7
1 9 1 7
   table=. 2 3$ word,'bat'
   $ table                         table
2 3                             saw
                                bat
```

The number of atoms in the shape of a noun is called its *rank*. Each position of the shape is called an *axis* of the array, and axes are referred to by indices 0, 1, 2, etc.

For example, axis 0 of `table` has length 2 and axis 1 has length 3. The last *k* axes of an array `b` determine *rank-k cells* or *k-cells* of `b`. For example, if `s=. 2 3 4` and

```
   b=.s$'abcdefghijklmnopqrstuvwx'
   b
abcd
efgh
ijkl

mnop
qrst
uvwx
```

then the list `abcd` is a 1-cell of `b`, and the letters are each 0-cells.

The rest of the shape vector is called the *frame* of `b` relative to the cells of rank *k*. Thus, if `$c` is `2 3 4 5`, then `c` has the frame `2 3` relative to cells of rank `2`, the frame `2 3 4 5` relative to 0-cells (atoms), and an empty frame relative to 4-cells.

A cell of rank one less than the rank of `b` is called an *item* of `b`; an atom has one item, itself. For example, the verb *from* (denoted by `{`) selects items from its right argument, as in:

```
   0{b            1{b          0{0{b
abcd           mnop         abcd
efgh           qrst
ijkl           uvwx
   2 1{0{b                     1{2{0{b              0{3
ijkl                         j                  3
efgh
```

Moreover, the verb *grade* (denoted by `/:`) provides indices to `{` that bring items to "lexical" order. Thus:

```
   g=./:n=.4 3$3 1 4 2 7 9 3 2 0
   n              g                g{n
3 1 4          1 0 3 2          2 7 9
2 7 9                           3 1 4
3 2 0                           3 1 4
3 1 4                           3 2 0
```

Negative numbers, as in _2-cell and _1-cell (an item), are also used to refer to cells whose *frames* are of the length indicated by the magnitude of the number. For example, the list `abcd` may be referred to either as a _2-cell or as a 1-cell of `b`.

Open and Boxed. The nouns discussed thus far are called *open*, to distinguish them from *boxed* nouns produced by the verb *box* denoted by <. The result of box is an atom, and boxed nouns are displayed in boxes. Box allows one to treat any array (such as the list of letters that represent a word) as a single entity, or atom. Thus:

```
   words=.(<'I'),(<'was'),(<'it')
   letters=. 'I was it'
   $words                       $letters
3                            8
   |.words                      |. letters
┌──┬───┬─┐                   ti saw I
│it│was│I│
└──┴───┴─┘
   2 3$words,|.words
┌──┬───┬──┐
│I │was│it│
├──┼───┼──┤
│it│was│I │
└──┴───┴──┘
```

B. VERBS

Monads and Dyads. Most verbs have two definitions, one for the *monadic* case (one argument), and one for the *dyadic* case.

The dyadic definition applies if the verb is preceded by a suitable left argument, that is, any noun that is not itself an argument of a conjunction; otherwise the monadic definition applies. The monadic case of a verb is also called a *monad*, and we speak of the *monad* `%` used in the phrase `%4`, and of the *dyad* `%` used in `3%4`.

Ranks of Verbs. The notion of verb rank is closely related to that of noun rank: a verb of rank *k* applies to each of the k-cells of its argument. For example (using the array `b` from Section A):

```
   ,b
abcdefghijklmnopqrstuvwx
   ,"2 b                  ,"_1 b
abcdefghijkl           abcdefghijkl
mnopqrstuvwx           mnopqrstuvwx
```

Since the verb *ravel* (denoted by `,`) applies to its entire argument, its rank is said to be *unbounded*. The *rank* conjunction `"` used in

the phrase `,"2` produces a related verb of rank 2 that ravels each of the 2-cells of `b` to produce a result of shape 2 by 12.

The shape of a result is the frame (relative to the cells to which the verb applies) catenated with the shape produced by applying the verb to the individual cells. Commonly these individual shapes agree, but if not, they are first brought to a common rank by adding leading unit axes to any of lower rank, and are then brought to a common shape by *padding* with an appropriate *fill* element: space for a character array, `0` for a numeric array, and a boxed empty list for a boxed array. For example if `s=. 2 3 4`:

```
   i."0 s        >'I';'was';'here'
0 1 0 0        I
0 1 2 0        was
0 1 2 3        here
```

The dyadic case of a verb has two ranks, governing the left and right arguments. For example:

```
   p=. 'abc'
   q=. 3 5$'wake read lamp '
   p,"0 1 q
awake
bread
clamp
```

Finally, each verb has three intrinsic ranks: monadic, left, and right. The definition of any verb need specify only its behaviour on cells of the intrinsic ranks, and the extension to arguments of higher rank occurs systematically. The ranks of a verb merely place upper limits on the ranks of the cells to which it applies, and its domain may include arguments of lower rank. Thus, matrix inverse (`%.`) has monadic rank 2, but treats degenerate cases of vector and scalar arguments as 1-column matrices.

Agreement. In the phrase `p v q`, the arguments of `v` must *agree* in the sense that their frames (relative to the ranks of `v`) must either match, or one must be a prefix of the other, as in `p,"0 1 q` above, and in the following examples:

```
   p," 1 1 q           3 4 5*i. 3 4          (i.3 4)*3 4 5
abcwake              0  3  6  9            0  3  6  9
abcread             16 20 24 28           16 20 24 28
abclamp             40 45 50 55           40 45 50 55
```

C. ADVERBS & CONJUNCTIONS

Unlike verbs, adverbs and conjunctions have fixed valence: an adverb is monadic (applying to a single argument to its *left*), and a conjunction is dyadic.

A conjunction applies to noun or verb arguments, and may produce as many as four distinct classes of results.

For example, `u&v` produces the *composition* of the verbs `u` and `v`; and `^&2` produces the *square* by combining the power function with the right argument `2`; and `2&^` produces the function *2-to-the-power*. The conjunction `&` may therefore be referred to by different names for the different cases, or it may be referred to by the single term *and* (or *with*), which roughly covers all cases.

D. COMPARATIVES

The comparison `x=y` is treated like the everyday use of equality (that is, with a reasonable relative tolerance), yielding `1` if the difference `x-y` falls relatively close to zero. Tolerant comparison also applies to other relations and to *floor* and *ceiling* (`<.` and `>.`); a precise definition is given in Part III under *equal* (`=`). An arbitrary tolerance `t` can be specified by using the *fit* conjunction (`!.`), as in `x =!.t y`.

E. PARSING & EXECUTION

A sentence is evaluated by executing its phrases in a sequence determined by the parsing rules of the language. For example, in the sentence `10%3+2`, the phrase `3+2` is evaluated first to obtain a result that is then used to divide `10`. In summary:

1. Execution proceeds from right to left, except that when a right parenthesis is encountered, the segment enclosed by it and its matching left parenthesis is executed, and its result replaces the entire segment and its enclosing parentheses.

2. Adverbs and conjunctions are executed before verbs; the phrase `,"2-a` is equivalent to `(,"2)-a`, not to `,"(2-a)` . Moreover, the left argument of an adverb or conjunction is the entire verb phrase that precedes it. Thus, in the phrase `+/ . */b`, the rightmost adverb `/` applies to the verb derived from the phrase `+/ . *`, not to the verb `*`.

3. A verb is applied dyadically if possible; that is, if preceded by a noun that is not itself the right argument of a conjunction.

4. Certain *trains* form verbs, adverbs, and conjunctions, as described in Section F.

5. To ensure that these summary parsing rules agree with the precise parsing rules prescribed below, it may be necessary to parenthesize any adverbial or conjunctival phrase that produces anything other than a noun or verb.

One important consequence of these rules is that in an unparenthesized expression the right argument of any verb is the result of the entire phrase to the right of it. The sentence `3*p%q^|r-5` can therefore be *read* from left to right: the overall result is `3` times the result of the remaining phrase, which is the quotient of `p` and the part following the `%`, and so on.

Parsing proceeds by moving successive elements (or their *values* in the case of pronouns and other names) from the tail end of a *queue* (initially the original sentence prefixed by a left marker `§`) to the front of a *stack*, and eventually executing some eligible portion of the stack and replacing it by the result of the execution.

For example, if `a=. 1 2 3`, then `b=.+/2*a` would be parsed and executed as follows:

```
§ b =. + / 2 * a
§ b =. + / 2 *           1  2  3
§ b =. + / 2           * 1  2  3
§ b =. + /           2 * 1  2  3

§ b =. +           / 2 * 1  2  3
§ b =. +               / 2  4  6
§ b =.               + / 2  4  6
§ b               =. + / 2  4  6

§ b                        =. 12
§                        b =. 12
§                             12
                            § 12
```

The foregoing illustrates two major points: 1) Execution of the phrase `2*1 2 3` is deferred until the next element (the /) is transferred; had it been a conjunction, the `2` would have been *its* argument, and the monad `*` would have applied to `1 2 3`; and 2) Whereas the *value* of the name `a` is moved to the stack, the name `b`

(because it precedes a copula) moves unchanged, and the pronoun `b` is assigned the value `12`.

The executions in the stack are confined to *the first four elements* only, and eligibility for execution is determined only by the *class* of each element (noun, verb, etc.), as prescribed in the following table:

					LEGEND
§=(	**v**	**n**	?	*Monad*	
§=(avn	v	**v**	**n**	*Monad*	a Adverb
§=(avn	**n**	**v**	**n**	*Dyad*	c Conjunction
§=(avn	**nv**	**a**	?	*Adverb*	n Noun
§=(avn	**nv**	**c**	**nv**	*Conj*	p Pronoun (name)
§=(avn	**v**	**v**	**v**	*Forkv*	v Verb
§=(	**v**	**v**	?	*Hookv*	§ Lmark
§=(	**ac**	**ac**	**ac**	*Forkc*	= Is
§=(	**ac**	**ac**	?	*Hookc*	(Lparen
§=(	**c**	**nv**	?	*Bond*	) Rparen
§=(	**nv**	**c**	?	*Bond*	? Any
np	**=**	**cavn**	?	*Is*	
(	**cavn**	**)**	?	*Punct*	
?	?	?	?	*Get Next*	

The classes of the first four elements of the stack are compared with the first four columns of the parse table, and the first row that agrees in all four columns is selected. The bold elements in the row are then subjected to the action prescribed in column 5, and are replaced by its result.

F. TRAINS

An isolated sequence (such as `(+ */)`) which the foregoing parsing rules do not resolve to a single part of speech is called a *train*, and may be further resolved as described below.

Meanings are assigned to certain trains of 2 or 3 elements and, by implication, to trains of any length by repeated resolution. For example, the trains `+-*%` and `+-*%^` are equivalent to `+(-*%)` and `+-(*%^)` :

a) A *verb* is produced by trains of three or two verbs, as defined by the diagrams:

```
        FORK                        HOOK
   g         g                  g        g
  / \       / \                / \      / \
 f   h     f   h              y   h    x   h
 |   |    / \ / \                 |        |
 y   y    x y x y                 y        y
```

For example, `5(+*-)3` is `(5+3)*(5-3)`. The ranks of the hook are infinite, and the ranks of the fork `f g h` are the maxima of corresponding ranks of `f` and `h`.

b) An *adverb* is produced according to the following definitions (using nv to denote noun or verb):

x (a1 a2)	*is*	x a1 a2
x (c nv)	*is*	x c nv
x (nv c)	*is*	nv c x
x(a c)	*is*	x a c x

For example, if `inv=. ^: _1`, then `^inv` is the inverse of `^`, that is, `^.` .

c) A *conjunction* is produced by any of the following definitions, in a manner analogous to the hooks and forks that produce verbs:

x (c1 v c2) y	*is*	(x c1 y) v (x c2 y)
x (a1 c2 a3) y	*is*	(x a1) c2 (y a3)
x (a1 c2 c3) y	*is*	(x a1) c2 (x c3 y)
x (c1 c2 a3) y	*is*	(x c1 y) c2 (y a3)
x (c1 c2 c3) y	*is*	(x c1 y) c2 (x c3 y)
x (c a) y	*is*	(x c y) a

Trains of two and three elements are called *bidents*, and *tridents*, respectively; hooks and forks are special cases. Tree displays illustrate the choice of the names fork and trident:

```
  mean=. +/ % #
  (tree=. 5!:4@<) 'mean'
  ┌─ / ── +
──┼─ %
  └─ #
```

```
  conj=. ]. , [.
  tree 'conj'
  ┌─ ].
──┼─ ,
  └─ [.
```

III. DEFINITIONS

Each main entry begins with the words being defined, and ends with the class. The ranks of each verb or derived verb are shown in parentheses, with unbounded rank denoted by _ , and with ranks dependent on the ranks of argument verbs shown as mu, lv, etc. Except for minor re-ordering to group related verbs, the order is that of the Summary Table of Appendix E, also shown in footers.

In defining conjunctions (and adverbs), `m` and `n` denote (left and right) noun arguments, and `u` and `v` denote verb arguments.

`=` (Verb)

SELF-CLASSIFY (_) `=y` classifies the items of the nub of `y` according to equality with the items of `y`, producing a boolean table of shape `#~.y` by `#y`. For example:

```
  y          ~.y        =y
abc        abc        1 0 1
def        def        0 1 0
abc
```

EQUAL (0 0) `x=y` is `1` if `x` is equal to `y`, and is otherwise `0`. The comparison is made with a *tolerance* `t`, normally `2` to the power `_44` but also controlled by the *fit* conjunction `!.`, as in `x =!.0 y`. Formally, `x=y` is `1` if `|x-y` does not exceed `t` times the larger of the magnitudes of `x` and `y`. Tolerance applies similarly to other verbs as indicated for each, notably to Match (`-:`), to Floor (`<.`), and to Signum(`*`), but not to Grade (`/:`).

`=.` `=:` (Copulas)

IS (local), IS (global) Used as in `a=.3` and `sum=. +/`. The copula `=.` is *local* as discussed under Explicit Definition (`:`), and `=:` is *global*. Copulas may also be used *indirectly*. For example, if `x=.'abc';'de'` and `(x)=.3 4;5 6 7` then `3 4` is assigned to the name `abc`, and `5 6 7` to `de`. The shapes of the names and the entity assigned must agree, as in `(2 2$'abcd')=. i. 2 2`.

`<` (Verb)

BOX (_) `<y` and `<!.n y` are *atomic encodings* of `y`; either has rank `0` and is decoded by `>`, and their *classes* (given by `>!._`) are `0` and `n`. See Section II.A.

LESS THAN (0 0) `x<y` is `1` if `x` is tolerantly less than `y`. See `=` .

`>` (Verb)

OPEN (0) Open is the inverse of box, that is, `y -: ><y`. When applied to an open array (that has no boxed elements), open has no effect. Opened atoms are brought to a common shape as discussed in Sec. II.B. The *class* of a boxed atom is given by `>!._`.
LARGER THAN (0 0) `x>y` is `1` if `x` is tolerantly larger than `y`. See `=`.

`<.` `>.` (Verbs)

FLOOR, CEILING (_) `<.y` gives the *floor* or *integer part* of `y`, and `<.y` is therefore the largest integer such that `(<.y) <: y`. The implied comparison with integers is tolerant. See Equal (`=`). The *ceiling* `>.y` is `-<.-y`. See McDonnell [10] for complex arguments.
LESSER OF, LARGER OF (0 0) `x<.y` is the lesser of `x` and `y`, and `x>.y` yields the larger. Thus, `3<.4 _4` is `3 _4`.

`<:` `>:` (Verbs)

DECREMENT, INCREMENT (_) `<:y` is `y-1` and `>:y` is `y+1`. (Also see `-.`)
LESS OR EQUAL, LARGER OR EQUAL (0 0) `x<:y` is `1` if `x` is less than or equal to `y`, and is otherwise `0`. See Equal (`=`).

`_` `_.` `_:` (Special, Noun, Verb)

NEGATIVE SIGN, INDETERMINATE, INFINITY (_) `_` followed by a digit denotes a negative number (as in `_3.4`), infinity (when used alone), or negative infinity (in `__`). It is also used in names. The *indeterminate* `_.` results from expressions such as `_-_` and `3+_.`. The verb `_:` yields infinity.

`+` `*` `-` `%` (Verbs)

CONJUGATE, SIGNUM, NEGATE, RECIPROCAL (_) The following definitions and examples apply (with `t` denoting tolerance as defined under Equal):

`+y`	`(|y*y)%y`	`+3j5` is `3j_5`
`*y`	`(y%|y)*t<:|y`	`*_3 0 5` is `_1 0 1`
`-y`	`0-y`	`-7` is `_7`
`%y`	`1%y`	`%4` is `0.25`

PLUS, TIMES, MINUS, DIVIDE (0 0) These are defined as in elementary arithmetic, but `0%0` is `0`. See McDonnell [11], and the resulting pattern in the function table `%/~@(i.@>:@+:-]) 3`.

`+.  *.  +:  *:` (Verbs)

REAL / IMAGINARY, POLAR, DOUBLE, SQUARE (_) `+.y` is `9 11 o.y` and `*.y` is `10 12 o.y` and `+:y` is `y+y` and `*:y` is `y*y`.

GREATEST COMMON DIVISOR (OR), LEAST COMMON MULTIPLE (AND), NOT-OR, NOT-AND (0 0) `x+.y` is the greatest common divisor of `x` and `y`, and `x*.y` is the least common multiple. For boolean arguments (`0` and `1`) `+.` is equivalent to *or*, and `*.` to *and*. Thus:

```
0 0 1 1 +. 0 1 0 1  is 0 1 1 1
0 0 1 1 *. 0 1 0 1  is 0 0 0 1
```

`x+:y` is `-.x+.y`, and `x*:y` is `-.x*.y`.

`-.` (Verb)

NOT (_) `-.y` is `1-y`; for a boolean argument it is the complement (not); for a probability, it is the complementary probability.

LESS (_ _) Items of `x-.y` are all of the items of `x` except for those that are cells of `y`.

`-:` (Verb)

HALVE (_) `-:y` is `y%2`.

MATCH (_ _) `-:` yields `1` if its arguments match: in shapes, boxing, and elements; but using tolerant comparison. See Equal (=).

`%.` (Verb)

MATRIX INVERSE (2) If `y` is a non-singular matrix, then `%.y` is the inverse of `y`, and hence `(%.y) +/ . *y` is the identity matrix `id=. =i.{:$y`.

More generally, `%.y` is defined in terms of the dyadic case as `id %. y`, or, equivalently, by the relation `(%.y)+/ . *x` is `(x%.y)`. The shape of `%.y` is `|.$y`. The degenerate vector and scalar cases are defined by using `,.y`, but the shape of the result is `$y`. For a non-zero vector `y`, the result of `%.y` is a vector collinear with `y` whose length is the reciprocal of that of `y`.

MATRIX DIVIDE (_ 2) If the columns of *y* are linearly independent, and if `#x` and `#y` agree, then `x%.y` minimizes the atoms of `r=.+/d*+d=.x-y+/ . *x%.y`. If `y` is square, it is necessarily invertible (since its columns are independent); the elements of `r` are

all `0`, and `y+/ . *x%.y` matches `x`. As in the monadic case, degenerate cases of `y` are treated as `,.y`.

Geometrically, `y+/ . *x%.y` is the projection of the vector `x` on the column space of `y`, the point in the space spanned by the columns that is nearest to `x`. Common uses of `%.` are in the solution of linear equations; and in the approximation of functions by polynomials, as in the expression `c=. (f x)%. x ^ / i.4`.

`%:` (Verb)

SQUARE ROOT (_) `%:y` is `2%:y`.

ROOT (0 0) `x %: y` is `y^%x`.

`^` `^.` (Verbs)

EXPONENTIAL, NATURAL LOGARITHM (_) `^y` is equivalent to `e^y`, where `e` is *Euler's* number `^1` (approximately `2.71828`). The *natural logarithm* (`^.`) is inverse to `^` (that is, `y= ^. ^ y` and `y= ^ ^. y`).

POWER, LOGARITHM (0 0) `x^2` and `x^3` and `x^0.5` are the *square*, *cube*, and *square root* of `x`. The general definition of `x^y` is `^y*^. x`, applying for complex numbers as well as real. For a non-negative integer right argument it is equivalent to `*/y#x`; in particular, `*/` on an empty list is `1`, and `x^0` is `1` for any `x`, including `0`.

The fit conjunction applies to the power as follows: `x ^!.k n` is `*/x-k*i.n`. In particular, `^!._1` is the *falling factorial* function.

The *base-x logarithm* `x^.y` is the inverse of power in the sense that `y=x ^. x ^ y` and `y=x ^ x ^. y`.

`u^:n` (Conjunction)

POWER (_) Two cases occur: a numeric integer `n`, and a gerund `n`.

Numeric case. The verb `u` is applied `n` times. For example, `-/^:2 y` is `-/-/y`. An array argument `n` may be used, as in `o.^: (i.4) 1`. Infinite `n` produces the limit of the application of `u`. Thus, `2&o.^:_ (1)` is `0.73908`, the solution of the equation y=Cos y.

If `n` is negative, the obverse `u^:_1` is applied `|n` times. The obverse (which is normally the inverse) is specified for five cases:

1. The pairs in the following lists:

```
< <: + +.        +: +~ - -. *.        *: *~ %
> >: + j./"1"_ -: -: - -. r./"1"_ %: %: %

%. ^  |. |: ,: ,~
%. ^. |. |: {. (<.@-:@#)&{.

;:                #.  ".  ;~
;&(' '&,&.>)"1    #:  ":  >@{.

/: \:     [ ]  o.        C.  j.     p.  r.
/: /:@|.  [ ]  %&(o.1) C.  %&0j1  p.  %&0j1@^.
```

2. Obviously invertible monads such as `-&3` and `10&^.` and `1 0 2&|:` and `3&|.` and `1&o.` and `a.&i.` as well as `u@v` and `u&v` and `u&.v` if `u` and `v` are invertible

3. Monads of the form `v/\` and `v/\.` where `v` is one of `+ * % = ~:`

4. Obverses specified by `:.`

5. All others by a linear approximation

Gerund case. (Compare with the gerund case of the adverb `}`)

```
x u^: (v0`v1`v2) y is (x v0 y) u^:(x v1 y) (x v2 y)
  u^: (v0`v1`v2) y is          u^:(  v1 y) (  v2 y)
  u^: (   v1`v2) y is          u^:(  v1 y) (  v2 y)
```

CHAIN (_) Denoting `x u^:n y` by Rn, it is the result of Rn–2 `u` Rn–1, where R0 is `x` and R1 is `y`. Also, `x u^:_ y` is Rn for the least n for which Rn equals Rn+1 and Rn+2 equals Rn+3. Thus, `|~^:_` is GCD. See Tu [12] .

`u^:v` (Conjunction)

POWER (_) `'u^:(v y.)y.':'' ^:_`
CHAIN (_ _) `'' : 'x.&u^:(x.&v)y.'`

`$` (Verb)

SHAPE OF (_) `$ y` yields the shape of `y` as defined in II.A.

SHAPE (1 _) The shape of `x$y` is `x,siy` where `siy` is the shape of an item of `y`:

```
   y              2 2 $ y
abcd           abcd
efgh           efgh
ijkl
               ijkl
               abcd
```

This example shows how the result is formed from the *items* of **y**, the last 1-cell (**abcd**) showing that the selection is cyclic.

$. (Pronoun)

SUITE See Explicit Definition (`:`) for use in sequence control.

$: (Proverb, Pro-adverb, Pro-conjunction)

SELF-REFERENCE (_ _ _) `$:` is a proxy that assumes the result of the phrase in which it occurs, the phrase being terminated on the left by a copula or the completion of the sentence. For example, ``1:`(*$:@<:)@.*5`` yields `!5`.

m~ (Adverb)

EVOKE (_) If `m` is a proverb, then `'m'~y` is equal to `m y`.

u~ (Adverb)

REFLEXIVE (_) `u~ y` is `y u y`.

PASSIVE (ru lu) `~` *commutes* or *crosses* connections to arguments: `x u~ y` is `y u x`.

~. (Verb)

NUB (_) `~.y` selects the *nub* of `y`, that is, all of its distinct items. For example:

```
   y          ~. y          ~. 3          $~. 3
ABC        ABC            3              1
ABC        DEF
DEF
```

More precisely, the nub is found by selecting the leading item, suppressing from the argument all items tolerantly equal to it, selecting the next remaining item, and so on.

~: (Verb)

NUBSIEVE (_) `~:y` is a boolean list `b` such that `b#y` is the nub of `y`.

NOT EQUAL (0 0) `x~:y` is `1` if `x` is tolerantly unequal to `y`. See Equal (=).

| (Verb)

MAGNITUDE (_) `|y` is `%:y*+y`. For example, `| _6 3j4` is `6 5`.

RESIDUE (0 0) The familiar use of residue is in determining the remainder on dividing a non-negative integer by a positive integer. For example, `3|0 1 2 3 4 5 6 7` is `0 1 2 0 1 2 0 1`. The definition `y-x*<. y % x+0=x` extends this notion to a zero left argument (which yields the right argument unchanged), and to negative and fractional arguments. For a negative left argument, the result ranges between the left argument and zero, as it does for a positive left argument. For example:

```
   _3 | _4 _3 _2 _1 0 1 2 3 4
_1 0 _2 _1 0 _2 _1 0 _2
   1 | 2.5 3.64 2 _1.6
0.5 0.64 0 0.4
```

However, to produce a true zero for cases such as `(%3)|(2%3)`, the residue is made tolerant: If `s=. y%x+x=0`, then `x | y` is `y-x*<.s` if `(x~:0)*.(>.s)~:<.s` and is otherwise `y*x=0`.

For example, `0.1 | 2.5 3.64 2 _1.6` is `0 0.04 0 0`. The definition also applies to complex numbers, using the properties of floor on complex arguments.

|. (Verb)

REVERSE, RIGHT SHIFT (_) `|.y` reverses the order of the items of `y`:

```
  y             |. y
abcd          ijkl
efgh          efgh
ijkl          abcd
```

The *right shift* `|.!.p y` is `_1|.!.p y`.

ROTATE, SHIFT (0 _) `x|.y` rotates the items of `y`:

```
  y             1 |. y          _1 |. y
abcd          efgh            ijkl
efgh          ijkl            abcd
ijkl          abcd            efgh
```

The phrase `x |.!.p y` produces a *shift*: the items `(-x){.x|.y` are amended by `(}.$y)($,)"1 _1 (|x)$r` where `r` is `p` unless `0=#p`, when it is the *fill* defined under `{.`.

|: (Verb)

TRANSPOSE (_) `|:y` reverses the order of the axes of `y`.

TRANSPOSE (1 _) `x|:y` moves axes `x` to the tail end. If `x` is boxed, the axes in each box are *run together* to produce a single axis in the result. For example:

```
   y              2 1|:y          (<2 1)|:y
abcd            aei             afk
efgh            bfj             mrw
ijkl            cgk
                dhl                |:i. 3 4
mnop                            0 4  8
qrst            mqu             1 5  9
uvwx            nrv             2 6 10
                osw             3 7 11
                ptx
```

u . v (Conjunction)

DETERMINANT (2) The phrases `-/ . *` and `+/ . *` are the determinant and permanent of square matrix arguments. More generally, `u . v` is defined in terms of a recursive expansion by minors along the first column. Thus, `u . v` is defined by:

```
v/@,`({."1 u . v$:@minors)@.(1&<@{:@$)"2
```

where `minors=. }."1@(1&([\.))` .

DOT PRODUCT (_ _) For vectors and matrices, `x+/ . *y` is equivalent to the *dot*, *inner*, or *matrix* product of math, and other rank-0 verbs such as `<.` and `*.` are treated analogously. In general, `u . v` is defined by `u@(v"(1+lv,_))` .

In other words, `u` is applied to the result of `v` on lists of "left argument cells" and the right argument *in toto*. The number of items in a list of left argument cells must agree with the number of items in the right argument. For example, if `v` has ranks `2` and `3` and the shapes of `x` and `y` are `2 3 4 5 6` and `4 7 8 9 10 11`, then there are `2 3` lists of left argument cells (each shaped `4 5 6`); and if the shape of a result cell is `sr`, then the overall shape is `2 3 7 8,sr`.

u .. v u .: v (Conjunctions)

EVEN, ODD

`u .. v` is `u -:@:+ u&v`

`u .: v` is `u -:@:- u&v`

m : n (Conjunction)

EXPLICIT DEFINITION (_ _ _) If `n` is non-numeric, the conjunc-

tion : produces a verb whose monadic and dyadic cases are determined by `m` and `n`, respectively. If `f=.'2 f y.' : 'y.^%x.'`, then `f 64` is the square root and `3 f 64` is the cube root of `64`.

As illustrated by the foregoing, `x.` and `y.` denote the arguments. In general, `m` and `n` are boxed lists, and the boxed sentences are executed in a sequence determined by the *suite* `$.`. An open list is treated as a single box, and the items of an open table are treated as a list of boxes. Thus:

```
   a=. ',x.[$.=.y.#1'
   b=. a;'x.=.(x.,0)+(0,x.)'
   g=. '1 g y.' : b

   1 g 4
1 4 6 4 1
   g 4
1 4 6 4 1
```

The suite `$.` is initially set to `i.ns`, where `ns` is the number of sentences; in this example, it is reset to execute sentence `1` the number of times specified by the right argument. The result of the verb is the result of the sentence executed last.

Any name assigned by the copula `=.` is made strictly local on its first assignment; that is, values assigned to the name have no effect on the use of the name outside of the verb or within other verbs invoked by it. The names `x.` `y.` `$.` `$:` are also local.

If sentence `k` begins with a name followed by a right parenthesis, the name is local and is set to `k}.i.ns`. Such a *label* is useful in setting the suite to effect *branching*.

`m : 1` yields an adverb; its left argument is substituted for `x.` in `m`.

`m : 2` is a conjunction.

The right arguments `20`, `21`, and `22` produce tacit definitions of verb, adverb, and conjunction, respectively. For example:

```
    (mean=. '(+/y.)%#y.' : 20)
┌───────────────┬─┬─────────┐
│┌─────────┬─┬─┐│%│┌─┬─┬─┐  │
││┌─┬─┐    │@│]││ ││#│@│]│  │
│││+│/│    │ │ ││ │└─┴─┴─┘  │
││└─┴─┘    │ │ ││ │         │
│└─────────┴─┴─┘│ │         │
└───────────────┴─┴─────────┘

   mean 1 2 3 4 5
3
```

```
   (each=. 'x.&.>' :21)              |. each 1 2 3;'baker'
+-+-+                             +-----+-----+
|&|>|                             |3 2 1|rekab|
+-+-+                             +-----+-----+

   (dot=. 'x./@(y."1 _)' : 22)          + dot *
+-+-+-------+                     +-----+-+---------+
|/|@|+-+---+|                     |+-+-+|@|+-+-+---+|
| | ||"|1 _||                     ||+|/|| ||*|"|1 _||
| | |+-+---+|                     |+-+-+| |+-+-+---+|
+-+-+-------+                     +-----+-+---------+

   1 2 3 + dot * i. 3 5
40 46 52 58 64
```

`m : v`   `u : n`   `u : v` (Conjunction)

EXPLICIT DEFINITION (_   _ _)   The first argument specifies the monadic case, and the second argument the dyadic case, using in `m` and `n` the conventions used in `m : n`.

`u :. v` (Conjunction)

OBVERSE (mu lu ru) The result of `u :. v` is the verb `u`, but with an assigned obverse `v` (used as the "inverse" under the conjunctions `&.` and `^:`).

`u :: v` (Conjunction)

ADVERSE The result of `u :: v` is that of `u`, provided that `u` completes without error; otherwise the result is the result of `v`.

`,` (Verb)

RAVEL (_) `,y` gives a list of the atoms of `y` in "normal" order; the result is ordered by items, by items within items, etc. The result shape is `1$*/$ y`. Thus, `,i.2 3 4` is equal to `i.*/2 3 4`.

APPEND ITEMS (_ _) `x,y` appends items of `y` to items of `x` after 1) Reshaping an atomic argument to the shape of the items of the other, 2) Bringing the items to a common rank (of at least `1`) by repeatedly *itemizing* (`,:`) any of lower rank, and 3) Bringing them to a common shape by padding with fill elements in the manner described in Section II.B. For example:

```
   'abc','d'           5 6 7,y=.i.2 3              y,7
abcd               5 6 7                       0 1 2
                   0 1 2                       3 4 5
                   3 4 5                       7 7 7
```

`,.` (Verb)

RAVEL ITEMS (_) If `y` is an atom, then `,.y` is `1 1$y`; otherwise, `,.y` is `,"_1 y`, the table formed by ravelling each item of `y`.

APPEND (_ _) `,.` is equivalent to `,"_1`.

`,:` (Verb)

ITEMIZE (_) `,:y` adds a single unit axis to `y`, making the shape `1,$y`.

LAMINATE (_ _) An atomic argument in `x,:y` is first reshaped to the shape of the other (or to a list if the other argument is also atomic); the results are then itemized and catenated, as in `(,:x),(,:y)` .

`;` (Verb)

RAZE (_) `;y` assembles along a leading axis the opened elements of the ravel of `y` :

```
   ]y=.;:'I sing of'                  ;y
+-+----+--+                        isingof
|I|sing|of|
+-+----+--+
```

LINK (_ _) `x;y` is `(<x),y` if `y` is boxed, and `(<x),<y` if `y` is open.

`u ;. n` (Conjunction)

CUT (_) The phrase `u;. 1 y` applies `u` to each of a set of intervals of items of `y` to produce the items of the result. Each interval begins at an occurrence of the *delimiter* `0{y`. For example:

```
   s=. 5 3
   <;.1 y=.' worlds on worlds'
+-------+---+-------+
| worlds| on| worlds|
+-------+---+-------+
```

```
 $;.1 y           s$i.9          +/;.1 s$i.9
7                0 1 2          9 12 15
3                3 4 5          3  5  7
7                6 7 8
                 0 1 2
                 3 4 5
```

The phrase `u;._1 y` differs only in that delimiters are excluded from the intervals. In `u;.2` and `u;._2` the delimiter is the *last* item, and marks the ends of intervals.

The phrase `u;.0 y` applies `u` to `y` after reversing `y` along each axis, and is equivalent to `(0 _1 */$y) u;.0 y`.

The monads `u;.3` and `u;._3` apply `u` to tessellation by "maximal cubes", that is, they are defined by their dyadic cases using the left argument `($$y)$<./$y`.

CUT (_ _) The dyads `u;.1` and `u;._1` and `u;.2` and `u;._2` differ from the monads in that the intervals are delimited by the ones in the boolean argument `x`. Thus:

```
   x=. 0 1 0 0 1 [ y=. i. 5 3

   y                   x+/;.1 y              x+/;.2 y
 0  1  2             18 21 24               3  5  7
 3  4  5             12 13 14              27 30 33
 6  7  8
 9 10 11
12 13 14
```

The phrase `x u;.0 y` applies `u` to a rectangle or cuboid of `y` with one vertex at the point in `y` indexed by `v=.0{x`, and with the opposite vertex determined as follows: the dimension is `|1{x`, but the rectangle extends *back* from `v` along any axis for which the index `k{v` is negative. Finally, the order of the selected items is reversed along each axis `k` for which `k{1{x` is negative. For example:

```
   (1 _2,:_2 3)+;.0 i. 4 5
11 12 13
6   7  8
```

The cases `u;.3` and `u;._3` provide (possibly overlapping) tessellations. The phrase `x u;._3 y` applies `u` to each *complete* rectangle of size `|1{x` obtained by beginning at all positions obtained as integer multiples of (each item of) the *movement* vector `0{x`. As in

`u;.0`, reversal of each piece occurs along an axis for which the dimension `1{x` is negative.

The degenerate case of a list `x` is equivalent to the left argument `1,:x`, and therefore provides a complete tessellation of size `x`.

The case `u;.3` differs from `u;._3` only in that any shards of sizes less than `|1{x` are included. For example:

```
   y              (3 2;2 3)<;.3 y
abcdef
ghijkl           +---+---+--+
mnopqr           |abc|cde|ef|
stuvwx           |ghi|ijk|kl|
yzABCD           +---+---+--+
                 |stu|uvw|wx|
                 |yzA|ABC|CD|
                 +---+---+--+
```

;: (Verb)

WORD FORMATION (1) `;:y` is the list of boxed words in the list `y` according to the rhematic rules of Section I.

(Verb)

TALLY (_) `#y` is the number of items in `y`.

COPY (1 _) If the arguments have an equal number of items, then `x#y` copies `+/x` items from *y*, with `i{x` repetitions of item `i{y`; if one is an atom, it is repeated to make the item count of the arguments equal; if both are atoms they are treated as one-element lists.

#. (Verb)

BASE-2 (1) The base-2 value of `y`, that is, `2#.y`.

BASE (1 1) `x#.y` is a weighted sum of the items of *y*; that is, `+/w*y`, where `w` is the product scan `*/\.}.x,1`. For example, if `a=.1 2 3[b=.24 60 60`, then `10#.a` is `123`, and `b#.a` is `3723`.

#: (Verb)

ANTIBASE-2 (_) `#: y` is the binary representation of `y`, and is equivalent to `(m#2)#:y`, where `m` is the maximum of the number of digits needed to represent the atoms of `y` in base `2`. For example:

```
   |:#:i.8
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
```

ANTIBASE (1 0) In simple cases, `#:` is inverse to `#.`; in general, `x#.x#:y` is `(*/x)|y`.

`!` (Verb)

FACTORIAL (_) For a non-negative integer argument `y`, the definition is `*/>:i.y`. In general, `!y` is `gamma >:y`.

OUT OF (COMBINATIONS) (0 0) For non-negative arguments `x!y` is the number of ways that `x` things can be chosen out of `y`. More generally, `(x!y)` is `(!y)%(!x)*(!y-x)` with the understanding that infinities (occasioned by `!` on a negative integer) cancel if they occur in both numerator and denominator. Thus:

```
     !/~3--~i.7
  1 _2  1  0  0  0  0
  0  1 _1  0  0  0  0
  0  1  0  0  0  0  0
  1  1  1  1  1  1  1
 _3 _2 _1  0  1  2  3
  6  3  1  0  0  1  3
_10 _4 _1  0  0  0  1
```

`!.` (Conjunction)

FIT (CUSTOMIZE) This conjunction modifies certain verbs in ways prescribed in their definitions. For example, `=!.t` is the relation of equality, using tolerance `t`.

`!:` (Conjunction)

FOREIGN: This conjunction is used to communicate with the host system as well as with the keyboard (as an input file) and with the screen (as an output file). Details are given in Appendix D.

`m/` `u/` (Adverb)

INSERT (_) If `m` is a gerund, then `m/y` inserts successive verbs from `m` between items of `y`. Thus, ``+`*/i.6`` is `0+1*2+3*4+5`. The gerund `m` may extend cyclically. For the verb case, `u/y` applies the dyad `u` between the items of `y`. Thus:

```
   i.3 2      +/i.3 2       */2 3 4
0 1        6 9           24
2 3
4 5
```

If `y` has no items (that is, `0=#y`), the result of `u/y` is the *identity element* of the function `u`. An identity element of a function `u` is a value `e` such that either `x u e` is `x`, or `e u x` is `x` for every `x` in the domain (or perhaps some significant sub-domain such as boolean) of `u`. This definition of insertion over an argument having *zero* items extends partitioning identities of the form `(+/y)-:(+/k{.y)+ +/k}.y` to the cases `k=. 0` and `k=. #y`.

The *identity function* of a verb `u` is a function `ifu` such that `(ifu y)-:(u/y)` if `0=#y`. The identity functions are:

```
   $&0@}.@$             <  >  + - +. ~: | (2 4 5 6 b.)
   $&1@}.@$             = <: >: * % *. %: ^ ! (1 9 11 13 b.)
   $&_@}.@$                   <.
   $&_ _@}.@$                 >.
   i.@(0&,)@(2&}.)@$          ,
   i.@(1&{.)@}.@$             C. {
   =@i.@(1&{.)@}.@$           %. +/ . *
   ifu@#                      u/
   $&(v^:_1 ifu$0)@}.@$       u&.v
```

FUNCTION TABLE (_ _) If `x` and `y` are numeric lists, then `x */ y` is their multiplication table. For example:

```
   1 2 3 */ 4 5 6 7
 4  5  6  7
 8 10 12 14
12 15 18 21
```

In general, each cell of `x` is applied to the entire `y`. Thus `x u/ y` is `x u"(lu,_) y`.

u/. (Adverb)

OBLIQUE (_) `u/.y` applies `u` to each of the oblique lines of a table `y`. For example, if `p=. 1 2 1` and `q=. 1 3 3 1`, then:

```
   p*/q          +//. p*/q         </.p*/q
1 3 3 1       1 5 10 10 5 1    ┌─┬───┬─────┬─────┬───┬─┐
2 6 6 2                        │1│3 2│3 6 1│1 6 3│2 3│1│
1 3 3 1                        └─┴───┴─────┴─────┴───┴─┘
```

More generally, `u/.y` is the result of applying `u` to the oblique lines of `_2`-cells of `y`. If the rank of `y` is less than two, `y` is treated as the table `,.y`.

KEY (_ _) `x u/. y` is `(=x) u@# y`, that is, items of `x` specify keys for corresponding items of `y` and `u` is applied to each collection of `y` having identical keys. Thus:

`1 2 3 1 3 2 1 </.'abcdefg'` is `'adg';'bf';'ce'` .

m\ (Adverb)

TRAIN (max over ranks of gerunds) `m\` is equivalent to the train of verbs represented by the gerund `m`.

u\ u\. (Adverb)

PREFIX, SUFFIX (_) `u\y` has `#y` items resulting from applying `u` to each of the prefixes `k{.y`, for `k` from `1` to `#y`. Thus, `<\'abc'` is `(,'a');'ab';'abc'`, and `+/\ 1 2 3 4` is `1 3 6 10`.

`u\.y` has `#y` items resulting from applying `u` to suffixes of `y`, beginning with one of length `#y` (that is, `y` itself), and continuing through a suffix of length `1`.

INFIX, OUTFIX (0 _) If `a` is positive, then the items of `a u\ y` result from applying `u` to each infix of length `a`. If `a` is negative, then `u` is applied to the non-overlapping infixes of length `|a`, including a possible final shard. For example, if `q=.'abcde'`, then `2<\q` is `'ab';'bc';'cd';'de'`, and `_2<\q` is `'ab';'cd';,'e'`.

If `x` is positive in the expression `x u\. y`, then `u` applies to the outfixes of `y` obtained by suppressing successive infixes of length `x`. If `x` is negative, the outfixes are determined by suppressing non-overlapping infixes, the last of which may be a shard: `2<\.q` is `'cde';'ade';'abe';'abc'`.

/: \: (Verbs)

GRADE UP, GRADE DOWN (_) `/:` *grades* its argument, yielding a permutation vector such that `(/: y) { y` sorts `y` in ascending order. For example:

```
   ]g=. /:y=. 3 1 4 2 1 3 3
1 4 3 0 5 6 2
   g{y
1 1 2 3 3 3 4
```

Elements of `g` that select equal elements of `y` are in ascending order.

If `y` is a table, `/:y` grades the base value of the rows, using a base

larger than twice the magnitude of any of the elements. Any `y` of higher rank is treated as `,.y`, that is, as if its items were each ravelled.

If `y` is literal, `/:y` grades according to the collating sequence specified by the alphabet `a.` ; any other collating sequence `cs` can be imposed by grading `cs i.y`.

Downgrade is like upgrade, except that the items of `(\:y){y` are in descending order.

SORT (_ _) `x/:y` is `(/:y){x`; that is, `x` is sorted into an order specified by `y`. In particular, `y/:y` (or `/:~y`) sorts `y`.

[ ] (Verbs)

SAME (_) Each yields its argument.

LEFT, RIGHT (_ _) `[` (*left bracket*) yields the left argument, and `]` the right.

[.]. (Conjunctions)

LEV, DEX `[.` yields the left argument and `].` the right.

{ (Verb)

CATALGUE (1) `{y` forms a catalogue from the atoms of its argument, its shape being the chain of the shapes of the opened items of `y`, and the common shape of the boxed results is `$y`. The case `{a;<b` is called the *Cartesian product* of `a` and `b`. Thus:

```
   c=. {'ht';'ao';'gtw'
   n=. { 10 11 ; i.2 3
   c                          n
+---+---+---+            +----+----+----+
|hag|hat|haw|            |10 0|10 1|10 2|
+---+---+---+            +----+----+----+
|hog|hot|how|            |10 3|10 4|10 5|
+---+---+---+            +----+----+----+

+---+---+---+            +----+----+----+
|tag|tat|taw|            |11 0|11 1|11 2|
+---+---+---+            +----+----+----+
|tog|tot|tow|            |11 3|11 4|11 5|
+---+---+---+            +----+----+----+
   $c                         $n
2 2 3                      2 2 3
```

FROM (0 _) If `x` is an integer in the range from `-#y` to `<:#y`, then `x{y` selects item `(#y)|x` from `y`. More generally, `x` may be a boxed list, whose successive elements are (possibly) boxed lists that specify selection along successive axes of `y`. For example:

```
   y                         2 0{y
abcdef                    mnopqr
ghijkl                    abcdef
mnopqr

   (<2 0){y                  (<2 0;1 3){y
m                         np
                          bd
```

Finally, if any `r=.>j{>x` used in the selection is itself boxed, selection is made by the indices along that axis that do not occur in `>r`. For example, `(<(<<2 0),(<<1 3)){y` is `gikl` .

m} u} (Adverb)

If `m` is numeric, the verb `m}` is `m"_}`

If `m` is a gerund, then:

```
x (v0`v1`v2)} y is  (x v0 y) (x v1 y)} (x v2 y)
  (v0`v1`v2)} y is           (   v1 y)} (   v2 y)
  (    v1`v2)} y is          (   v1 y)} (   v2 y)
```

ITEM AMEND (_) `u}y` is an amendment of the items of `y` whose shape is the shape of an item of `y`. Each atom is selected as a corresponding atom in an item, the item being specified by the index in the corresponding position of `u y` (whose shape must be the shape of an item of `y`). If `m=.0 1 3 1 2`, and `u=. m&[`, then:

```
  y              u y            u}y            m}y
abcde          0 1 3 1 2      agrio          agrio
fghij
klmno
pqrst
```

AMEND (_ _) If `x` has the same shape as the index array `j=. x u y`, then `x u} y` amends `x` and `y` by inserting atoms of `x` in the selected positions of `y`. In other words, the values of `,x` replace the values of `(,j){,y`.

More generally, the shape of `x` may be a suffix of the shape of `j`, and the result is an amendment of `y` by `($j)$,x`.

Thus, `'BD'  1 3} 'abcd'` is `aBcD`, and if `x=.'AGMS '` and `u=.(<0 1)&|:@i.@$@]` (selecting diagonal indices), then `u y` is `0 6 12 18`, and:

```
  y              x u} y          'Z' u} y
abcde          Abcde            Zbcde
fghij          fGhij            fZhij
klmno          klMno            klZno
pqrst          pqrSt            pqrZs
```

{. (Verb)

HEAD (_) `{.y` selects the leading item of `y`; that is, `0{y`.

TAKE (1 _) If `x` is an atom, `x{.y` takes from `y` an interval of `|x` items; beginning at the front if `x>:0`, ending at the tail if `x<0`:

```
  y     2{.y      4{.y     _4{.y
0 1    0 1       0 1      0 0
2 3    2 3       2 3      0 1
4 5              4 5      2 3
                 0 0      4 5
```

In an *overtake* (as in `4{.y` and `_4{y` above), extra items consist of *fills*; zeros if `y` is numeric, `<$0` if it is boxed, and spaces otherwise. The fill `f` is also determined by *fit*, as in `{. !. f`.

If `y` is an atom, the result of `1{.y` is a one-element list. Finally, if `$y` is `0,s`, then the fill items are `s$0`. In general, `x` may be a list of length not more than `$$y`; the effect of element `k` is `(k{x){."((${y)-k) y`.

}. (Verb)

BEHEAD (_) `}.y` selects the rest of the items of `y` after the first; it is equivalent to `1}.y`.

DROP (0 _) `x }. y` drops (at most) `|x` items from `y` in a manner analogous to `{.`.

}: {: (Verbs)

TAIL, CURTAIL (_) `}:y` is `_1}.y` and `{:y` is `_1{y`.

m " n (Conjunction)

CONSTANT (`|.3$|.n`) The derived verb `m"n` produces the constant result `m` for each cell to which it applies. If `n` has one element, it specifies all ranks; if two, the last of them specifies the rank of the monad as well as the right rank.

u " n (Conjunction)

RANK (|.3$|.n) The verb `u"n` is equal to `u` except that its ranks are determined by `n` as specified under `m"n`. See Section II.B.

u " v (Conjunction)

RANK (mv lv rv) The verb `u"v` is `u` but with the ranks of `v`. Use `b.` to obtain ranks, as in `v b. 0` and `u"v b. 0`.

". (Verb)

DO (1) `".y` is the result of executing the character list or atom `y`. Thus, `".'a=. 3+4'` assigns the value `7` to `a` and yields the explicit result `7`. The result of `".` on the empty list is itself.

DO LEFT IF ERROR (1 1) `x ". y` is the same as `".y` if `y` is a valid sentence; otherwise it is the result of `".x`, which may invoke an error report in the normal manner.

": (Verb)

FORMAT (_) `": y` is equal to `x ": y`, where `x` is chosen to provide a minimum of one space between columns. Default output is identical to this monadic case.

The fit conjunction (`!.`) specifies the number of digits for real numbers. Thus, `":!. 4 (5%3)` yields `1.667`.

FORMAT (1 1) `x ": y` produces a literal representation of `y` in a format specified by `x`. Each element `e` of `x` controls the representation of the corresponding element of `y` as follows:

`w=. <.|e` specifies the total width allocated; if this space is inadequate, the entire space is filled with asterisks. If `w` is zero, enough space is allocated.

`d=.<.10*(|e)-w` specifies the number of digits following the decimal point (which is itself included only if `d` is not zero).

Any negative sign is placed just before the leading digit.

If `e>:0`, the result is right-justified in the space `w`.

If `e<0`, the result is put in exponential form (with one digit before the decimal point) and is left-justified except for two fixed spaces allowed on the left (including one for a possible negative sign).

m ` n   m ` v   u ` n   u ` v (Conjunction)

GERUND: `u`v` is `au,av`, where `au` and `av` are the (boxed noun) *atomic representations* of `u` and `v`. Moreover, `m`n` is `m,n` and `m`v` is `m,av` and `u`n` is `au,n`. See Bernecky [13].

m ` : n (Conjunction)

EVOKE GERUND: The cases `` `:3 `` and `` `:6 `` correspond to the adverbs / (*Insert*), and \ (*Train*). The remaining cases follow:

m ` : 0

APPEND (max over ranks of gerunds) The items resulting from the verb `` m`:0 `` are the results of the individual gerunds in `m`.

u @ v (Conjunction)

ATOP (mv) `u@v y` is `u v y` (the same as `u&v y`).

ATOP (lv rv) `x u@v y` is `u x v y`. For example, `3 |@- 7` is `4`.

m @. v (Conjunction)

AGENDA (mv, lv, rv) `m@.v` is a verb defined by the gerund `m` with an *agenda* specified by `v`; that is, if the result of the verb `v` is the index `i`, then the verb represented by element `i` of `m` is applied.

u @: v (Conjunction)

AT (_ _ _) `@:` is equivalent to `@` except that the ranks are infinite.

m & v   u & n (Conjunctions)

WITH (_) `m&v y` is defined by `m v y`, and `u&n y` by `y u n`.

u & v (Conjunction)

COMPOSE or AND (mv mv mv) `u&v y` is `u v y` (the same as `u@v y`) and `x u&v y` is `(v x) u (v y)`.

u &. v (Conjunction)

UNDER (mv mv mv) The verb `u &.v` is equivalent to the composition `u & v` except that the verb obverse to `v` is applied to the result for each cell. Obverses are discussed under the power conjunction `^:`. For example, the phrase `x+&.^. y` yields the product of `x` and `y`, and `|.&.> y` reverses each of the boxed elements of `y`.

u &: v (Conjunction)

APPOSE (_ _ _) &: is equivalent to & except that its ranks are infinite.

? (Verb)

ROLL (_) ? yields a uniform random selection from the population i.y.

DEAL (0 0) x?y is a list of x items randomly chosen without repetition from i.y.

) (Special)

LABEL The parenthesis sets off a *label* in explicit definition (see :).

a. (Noun)

ALPHABET a. is a list of the elements of the alphabet; it determines the *collating sequence* used in grading and sorting (/: and \:).

A. (Verb)

ATOMIC PERMUTE (1 0 _) If T is the table of all !n permutations of order n arranged in lexical order (that is, /:T is i.!n), then k is said to be the *atomic representation* of the permutation k{T. Moreover, k A. b permutes items of b by the permutation of order #b whose atomic representation is (#b)|k. For example, 1 A. b transposes the last two items of b and _1 A. b reverses all items, and 3 A. b and 4 A. b rotate the last three items of b. Finally, the phrase (i.!n) A. i.n produces the ordered table of all permutations of order n, as does (i.&! A. i.) n.

The monad A. applied to any cycle or direct permutation yields its atomic representation. Thus, A. 0 3 2 1 is 5, as are A.3 2 1 and A.0;2;3 1 and A.<3 1.

b. (Adverb)

BOOLEAN (_) m b. y is 0 m b.y.

BOOLEAN (0 0) For m e.i.16, the phrase m b. yields the *m*th boolean function, that is, x m b. y is the value in row x and column y of 2 2$(4#2)#: m. For example, 7 b. is

or and `1 b.` is *and*. The integer `m` may also be negative (down to `_16`), and is treated as `16|m`.

BASE CHARACTERISTICS (_) `u b. y` gives the obverse of `u` if `y=. _1`; its ranks if `y=. 0`; and its identity function if `y=. 1`. Thus:

```
   ^ b. _1
^.
   ^ b. 0
_ 0 0
   ^ b. 1
$&1@().@$)
```

c. (Verb)

CHARACTERISTIC (2) `c. y` yields the *characteristic, own,* or *eigen* values of its argument, arranged in ascending order on imaginary part within real within magnitude. An atom or list `y` is treated as the table `,.y`.

CHARACTERISTIC (0 2) `0 c. y` is a diagonal matrix with the eigenvalues `c. y` on the diagonal. Also, `_1 c. y` and `1 c. y` are the left and right eigenvectors. Thus, `+/ . */ _1 0 1 c. y` equals `y`.

C. (Verb)

PERMUTE [CYCLE TO/FROM DIRECT] (1) If `p` is a permutation of the atoms of `i.n`, then `p` is said to be a permutation vector of order `n`, and if `n=#b`, then `p{b` is a permutation of the items of `b`.

`C.p` yields a list of boxed lists of the atoms of `i.#p`, called the *standard cycle representation* of the permutation `p`. For example, if `p=.4 5 2 1 0 3`, then `C.p` is `(,2);4 0;5 3 1` because the permutation `p` moves to position `2` the item `2`, to `4` the item `0`, to `0` the item `4`, to `5` the item `3`, to `3` the item `1`, and to `1` the item `5`. The monad `C.` is self-inverse; applied to a standard cycle it gives the corresponding direct representation.

A given permutation could be represented by cycles in a variety of ways; the standard form is made unique by the following restrictions: the cycles are disjoint and exhaustive (i.e., the atoms of the boxed elements together form a permutation vector); each boxed cycle is rotated to begin with its largest element ; and the boxed cycles are put in ascending order on their leading elements.

PERMUTE (1 _) If `p` and `c` are standard and cycle representations of order `#b`, then `p C.b` and `c C.b` produce the corresponding permutations of items of `b`. More generally, since the tally of `b` determines the order of the permutation, the arguments `p` and `c` can be *non-standard* in ways to be defined. In particular, negative integers down to `-#b` may be used, and are treated as their residues modulo `#b`.

If `q` is not boxed, and if the elements of `(#b) |q` are distinct, then `q C.b` is equivalent to `p C.b`, where `p` is the standard form of `q` given by `p=. ((i.n)-.n|q),n|q` where `n=.#b`. In other words, positions occurring in `q` are moved to the tail end. If `q` is boxed, the elements of `(#b) |>j{q` must be distinct for each `j`, and the boxes are applied in succession: `(2 1;3 0 1) C.i.5` is `(<2 1) C.(<3 0 1)C.i.5`, and is equivalent to the standard direct permutation `1 2 3 0 4`.

The monad `C.` is extended to non-negative non-standard cases by treating any argument `q` as a representation of a permutation of order `1+>./; q`.

D. (Conjunction)

DERIVATIVE (mu) `u D. n` is the `n`th derivative of `u`. Thus:

```
   cube=. ^&3"0
   cube D. 1 x=. 2 3 4
12 27 48
   cube D. 2 x
12 18 24
   cube D. 3 x
6 6 6
```

If the argument rank of `u` is `a` and the result rank is `r`, then the argument rank of `u D. n` is also `a`, but its result rank is `r+a`: the result of `u D. n` is the (`n`th) derivative of each atom of the result of `u` with respect to each element of its argument. For example:

```
   (volume=. */"1) x
24
   volume D. 1 x
12 8 6
   (VOLUMES=. */\"1) x
2 6 24
```

```
   VOLUMES D. 1 x
1 3 12
0 2  8
0 0  6
   determinant=. -/ . * [. permanent=. +/ . *
   ]m=.*:i.3 3    determinant D.1 m    permanent D.1 m
 0  1  4         _201  324 _135        2249 1476 1017
 9 16 25          132 _144   36         260  144   36
36 49 64          _39   36   _9          89   36    9
```

The following adverbs first assign ranks to their arguments, and then take the first derivative; they are convenient for use in scalar and vector calculus. Thus, if `y=. 1r2p1 _1r4p1` and `sin=. 1&o.`:

```
    D=. ("0) (D.1)
   VD=. ("1) (D.1)
   */\ VD x              sin y                 sin D D y
1 3 12              1 _0.707107           _1 0.707107
0 2  8
0 0  6
```

e. (Verb)

RAZE IN (_) `e.y` produces a boolean result that determines for each atom of `y` whether its open contains each item of the raze of `y`. Thus if `y=.'abc';'dc';'a'`, then:

```
   y                        ;y              e.y
┌───┬──┬─┐               abcdca           1 1 1 0 1 1
│abc│dc│a│                                0 0 1 1 1 0
└───┴──┴─┘                                1 0 0 0 0 1
```

MEMBER (IN) (_ _) If `x` has the shape of an item of `y`, then `x e. y` is `1` if `x` matches an item of `y`. In general, `x e. y` is `(#y)>y i. x`. Thus:

```
   'cat' e. 'abcd'                  'cat' e. 2 3$ 'catdog'
1 1 0                             1
```

E. (Verb)

MEMBER OF INTERVAL (_ _) The *ones* in `b=. x E. y` indicate the beginning points of occurrences of the pattern `x` in `y`:

```
   'co' E. 'cocoa'
1 0 1 0 0
```

```
   (0 1 +/ i. 3) E. 4 | +/~ i. 5
1 0 0 0 0
0 0 0 0 0
0 0 1 0 0
0 1 0 0 0
0 0 0 0 0
```

`f.` (Adverb)

FIX If `sum=.+/` and `g=.sum f.\` then the verb `sum` is *fixed* in the definition of `g`, in that subsequent changes in the definition of `sum` will not affect the definition of `g`. The name denoted by `m` is (recursively) replaced by its referent.

`i.` (Verb)

INTEGERS (1) The shape of `i.y` is `|y`, and its atoms are the first `*/|y` non-negative integers. For example:

```
   i. 2 3              i. 2 _3
0 1 2               2 1 0
3 4 5               5 4 3
   i. ''               i. 4                  i. _4
0                   0 1 2 3               3 2 1 0
```

As shown in the examples, a negative element in `y` causes reversal of the atoms along the corresponding axis.

INDEX OF (_ _) If `rix` is the rank of an item of `x`, then the shape of the result of `x i.y` is `(-rix)}.$y`. Each atom of the result is either `#x` or the index of the first occurrence among the items of `x` of the corresponding `rix`-cell of `y`. The comparison in `x i. y` is tolerant.

`j.` (Verb)

COMPLEX (_ 0 0) `j.y` is `0j1*y` and `x j. y` is `x+j. y`.

`NB.` (Special)

COMMENT The rest of the line is ignored.

`o.` (Verb)

PI TIMES (_) `o. y` yields *pi* times `y`. For example, `o. 1` is approximately `3.14159`.

CIRCLE (0 0) If `k>:0`, then `k o.y` yields one of the circular, hyperbolic, or pythagorean functions, as follows:

k	Function	*k*	Function	*k*	Function
0	`%:1-y^2`	4	`%:1+y^2`	8	`%:->:y^2`
1	`Sine y`	5	`Sinh y`	9	`(y++y)%2`
2	`Cosine y`	6	`Cosh y`	10	`|y`
3	`Tangent y`	7	`Tanh y`	11	`(y-+y)%0j2`
				12	`(^.*y)%0j1`

`(-k)o.y` is inverse to `k o. y`. The cases `_1`, `_4`, `_9`, and `_10` are `Arcsine y` and `%:<:y^2` and `y` and `+y`. The arguments of sin, cos, and tan (and the *results* of their inverses) are in radians.

`p.` (Verb)

POLYNOMIAL (1) `p.` is a self-inverse transformation between an open list of coefficients of a polynomial and the corresponding boxed list of a multiplier and a list of roots: the functions `y&p.` and `(p. y)&p.` are equivalent.

POLYNOMIAL (1 0) If `x` is open, then `x p.y` is the result of the polynomial in `y` with coefficients `x`; that is, `+/x*y^i.#x`. If `x` is boxed, then `x p y` is the polynomial in terms of a multiplier `>{.x` and roots `>{:x`; that is, `(>{.x)**/y-(>{:x)`.

`r.` (Verb)

POLAR (_ 0 0) `r.y` is `^j.y` and `x r. y` is `x*r. y`.

`x.` `y.` (Surrogate arguments)

These denote the arguments in an explicit definition, using `:`.

`0:` `1:` to `9:` (Verbs)

ZERO, ONE, ... , NINE: (_ _ _) The results are `0` and `1` etc.

REFERENCES

1. Iverson, K.E., *Tangible Math*, ISI, 1991.
2. Iverson, K.E., *Arithmetic*, ISI, 1991.
3. Iverson, K.E., *Programming in J*, ISI, 1992.
4. McIntyre, D.B., *Mastering J*, APL Conference 1991, ACM.
5. McIntyre, D.B., *Language as an Intellectual Tool: From Hieroglyphics to APL*, IBM Systems Journal, December, 1991.
6. Hui, R.K.W., *An Implementation of J*, ISI, 1992.
7. Falkoff, A. D., and K. E. Iverson, *The Design of APL*, IBM Journal of R&D, July 1973 and *The Evolution of APL*, ACM Sigplan Notices, August 1978.
8. Hui, et al., *APL\?*, APL90, ACM.
9. Iverson, K. E., *A Dictionary of APL*, ACM APL Quote-Quad, September, 1987.
10. McDonnell, E.E., *Complex Floor*, APL73, ACM.
11. McDonnell, E.E., *Zero Divided by Zero*, APL76, ACM.
12. Tu, Hai-Chen, and A.J. Perlis, IEEE Software, January 1986.
13. Bernecky, Robert, and R.K.W. Hui, *Gerunds and Representations*, APL91, ACM.

ACKNOWLEDGMENT

I am indebted to Mr. Roger Hui for his rapid development of a flexible, reliable, and highly portable implementation of J, for his close collaboration in the design of the language, for critical reading of successive drafts of the dictionary, and for significant contributions to the treatments of certain topics, in particular, the identity functions.

APPENDIX A

ALTERNATIVE SPELLINGS FOR NATIONAL USE CHARACTERS

		.	:			.	:
@	AT.	AT1.	AT2.	#	NO.	NO1.	NO2.
\	BS.	BS1.	BS2.	]	RB.	RB1.	RB2.
^	CA.	CA1.	CA2.	}	RC.	RC1.	RC2.
`	GR.	GR1.	GR2.	$	SH.	SH1.	SH2.
[	LB.	LB1.	LB2.	|	ST.	ST1.	ST2.
{	LC.	LC1.	LC2.	~	TI.	TI1.	TI2.

APPENDIX B
NUMERIC CONSTANTS

NOUNS

The symbols used in forming numeric constants are interpreted in a sequence determined by the following hierarchy:

`.`	The decimal point is obeyed first
`_`	The negative sign is obeyed next
`e`	Exponential (scientific) notation
`r`	Rational number
`ad ar j`	Complex (magnitude and angle) in degrees or radians; Complex number
`p x`	Numbers based on pi (`o.1`) and on Euler's number (the exponential `^1`)
`b`	Base value (using `a` to `z` for `10` to `35`)

For example, `2.3` denotes two and three-tenths and `_2.3` denotes its negation; but `_2j3` denotes a complex number with real part `_2` and imaginary part `3`, *not* the negation of the complex number `2j3`. Furthermore, symbols at the same level of the hierarchy cannot be used together: `1p2x3` is an ill-formed number.

The following lists illustrate the main points:

```
   2.3e2 2.3e_2 2j3 2r3
230 0.023 2j3 0.666667
   2p1 1r2p1 1r4p1 1p_1
6.28319 1.5708 0.785398 0.31831
   1x2 2x1 1x_1
7.38906 5.43656 0.367879
   2e2r4j2e2r2 2e2r4j2e2r2p1 2ad45 2ar0.785398
50j100 157.08j314.159 1.41421j1.41421 1.41421j1.41421
   16b1f 10b23 _10b23 1r10b23 1e2b23 2b111.111
31 23 _17 3.2 203 7.875
```

VERBS

A single digit followed by a colon denotes the corresponding constant verb of infinite rank. For example, if `x=. 1 2 3 4`, then:

```
   0: x                 0:"0 x             9: x
0                    0 0 0 0             9
   8:"1 i. 3 4          3 7: 4
8 8 8                7
```

APPENDIX C

LOCATIVES

A name that includes an underbar (_) is a *locative*. Names used in a *locale* `F` can be referred to in another locale `G` by using the prefix `F` in a locative name of the form `F_pqr`, thus avoiding conflict with otherwise identical names in the locale `G`.

The referent of a locative can be established in either of two ways:

a) By assignment, as in `F_pqr=. i. 5`.

b) By *saving* a session in some locale; the names established in the session can thereafter be referred to by using the locale as a prefix in a locative name. For example:

```
   names=. 4!:1
   copy=. 2!:4
   save=. 2!:2
   save <'TOOLS'
1
   4!:55 names 3
1 1 1
   TOOLS_copy <'TOOLS'
1
   names 3
┌────┬─────┬────┐
│copy│names│save│
└────┴─────┴────┘
```

Appendix D: FOREIGN CONJUNCTION

[x] optional   Names boxed, as in 0!:2<'inp'

0!:0 y The list y is executed by the host system, and the result is returned

0!:1 y Like 0!:0, but yields '' without waiting for the host to finish

[x] 0!:2 y A script (file) input is chosen by y except that the value <'' chooses the keyboard; the resulting execution log is appended to file x

0!:2 <'profile.js' is executed at the start of every session

[x] 0!:3 y is like 0!:2, but execution log is not screened

[x] 0!:4 y is like 0!:2, but the lines in y are separated by (system-

[x] 0!:5 y dependent) new-line characters. Case 5 does not screen the log

0!:55 y (Or Control D)Terminate session

1!:0 y Directory

1!:1 y File read: y is a file to be read; result is a string of file contents; y may be a boxed name suited to the host file system, or the number 1, for keyboard as the source file

x 1!:2 y File write: x is a string; y is a boxed file name, or 2 for screen output

x 1!:3 y File append: like x 1!:2 y, but appends rather than replaces

1!:4 y File size: y is a boxed file name

1!:11 y Indexed file read: y is a list of a boxed file name and a boxed index and length. The index may be negative. If the length is elided, the read goes to the end

x 1!:12 y Indexed file write: Like indexed read, with x specifying the list to be written. The file positions must already exist

1!:55 y File erase: y is a file name

x 2!:0 y Name class (as in 4!:0) of x in locale (file) y

2!:1 y List of names in locale y

[x] 2!:2 y Save global names in locale y

[x] 2!:3 y Protected save of locale

[x] 2!:4 y Copy object x from locale y

[x] 2!:5 y Protected copy from locale y

x 2!:55 y Erase object x from locale y

3!:0 y Storage type of the noun y, encoded as 2^i.6 for boolean, literal, integer, floating, complex, boxed

3!:1 y Internal representation of noun y

3!:2 y Convert from internal

4!:0 y Name class of boxed name: **i.7** for undefined (**_1** if not valid), not used, noun, verb, adverb, conjunction, other

[x] 4!:1 y Name list: result is a list of boxed names belonging to the classes 1 to 5 (see **4!:0**) in **y**. The optional left argument lists the initial letters of names to be included

4!:55 y Erase name **y** **4!:56 y** Erase *given* name **y**

x 5!:0 Fix **5!:0** is an adverb yielding the inverses of **5!:1** and **5!:3**

5!:n y Representation of name **y**:

1. Atomic
2. Display
3. Workspace Interchange Standard
4. Tree
5. Linear

6!:0 y Time stamp: in order YMDHMS. +

6!:1 y Time since start of session

[x] 6!:2 y Seconds to execute sentence **y** averaged over **x** times (default **1**)

6!:3 y Delay for **y** seconds

7!:0 y Space currently in use

7!:1 y Space used since start of session

7!:2 y Space to execute sentence **y**

(Except as noted, **8!:** applies only to PCs) **8!:0 y** Query CGA mode

8!:1 y Set non-CGA if **y=.0**; CGA if **1**

8!:4 y Query screen attributes (**4** by **2** table of digits **0** to **15** as in DOS)

8!:5 y Set screen attributes

8!:7 y Refresh screen

8!:9 y Applies editor to **y**, a string with lines delimited by the line-feed (**10{a.**). Press F1 for definition of function keys

Query	Set
8!:16 y (MAC) font attributes	**8!:17 y**
8!:19 y Print screen (MAC)	
9!:0 y Random link	**9!:1 y**
9!:2 y Default display forms (see **5!:n**)	**9!:3 y**
9!:4 y Input prompt	**9!:5 y**
9!:6 y Box-drawing characters	**9!:7 y**
9!:8 y Error messages	**9!:9 y**

11!:0 y (Windows) WP driver . Press F1 for description

11!:1 y (Windows) Edit the WP string **y**

128!:0 y QR decomposition of **y**. Result is **a;b**, where **a** is orthonormal (**+|:a** is **a** inverse), and **a +/ . * b** is **y**.

128!:1 Invert square upper triangle

		.	:	Page
=	Self-Classify • Equal	Is (Local)	Is (Global)	71
<	Box • Less Than	Floor • Lesser of	Decrem • Less Or Eq	
>	Open • Larger Than	Ceiling • Larger of	Increm • Larg Or Eq	72
_	Negative Sign /*Infinity*	*Indeterminate*	Infinity	
+	Conjugate • Plus	Real / Imag • GCD (Or)	Double • Not-Or	73
*	Signum • Times	Polar • LCM (And)	Square • Not-And	
-	Negate • Minus	Not (1–) • Less	Halve • Match	
%	Reciprocal • Divide	Matrix Inv • Mat Divide	Square Root • Root	74
^	Exponential • Power	Natural Log • Log	**Power • Chain**	75
$	Shape Of • Shape	Suite	Self-Reference	76
~	***Reflex • Pass*** • EVOKE	Nub •	Nub Sieve • Not-Eq	
|	Magnitude • Residue	Reverse • Rotate (Shift)	Transpose	77
.	**Det • Dot Product**	**Even**	**Odd**	78
:	**Explicit Definition**	**Obverse**	**Adverse**	79
,	Ravel • Append Items	Ravel Items • Append	Itemize • Laminate	80
;	Raze • Link	**Cut**	Word Formation •	81
#	Tally • Copy	Base 2 • Base	Antibase 2 • Antib	82
!	Factorial • Out Of	**Fit (Customize)**	**Foreign**	83
/	***Insert • Table*** • INSERT	***Oblique • Key***	Grade Up • Sort	84
\	***Prefix • Infix*** • TRAIN	***Suffix • Outfix***	Grade Down • Sort	85
[	Same • Left	**Lev**		
]	Same • Right	**Dex**		
{	Catalogue • From	Head • Take	Tail •	86
}	***Amend***	Behead • Drop	Curtail •	87
"	**Rank** • CONSTANT	Do • Do left if error	Format	89
`	**Tie (Gerund)**		**Evoke Gerund**	90
@	**Atop**	**Agenda**	**At**	
&	**Bond /Compose**	**Under (Dual)**	**Appose**	91
?	Roll • Deal			
)	Label	a. *Alphabet*	A. Atomic Permute	
b.	***Boolean***	c. Characteristic	C. Cyc-Dir • Perm	92
D.	**Derivative**	e. Raze In • Member (In)	E. • Member of Int	93
f.	***Fix***	i. Integers • Index Of	j. Imagin • Cmplx	95
NB.	Comment	o. Pi Times • Circle	p. Polynomial	96
r.	Angle • Complex	x. Left Argument	y. Right Argument	
0:	Zero 1: One (to 9:)			

Appendix E: VOCABULARY